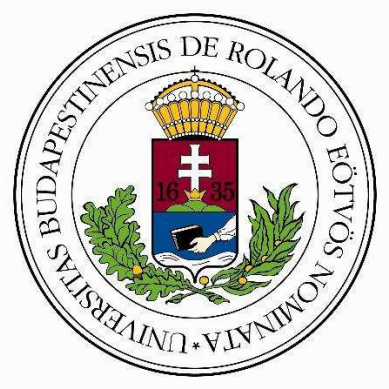


Eötvös Loránd Tudományegyetem

Informatika Kar

Programozási Nyelvek És

Fordítóprogramok Tanszék

Útkeresési Algoritmus Tanító Alkalmazás

Pataki Norbert

Adjunktus, PhD

Hetei Noel

Programtervező Informatikus, BSc

Budapest, 2019

Tartalomjegyzék

1. Bevezetés

Az elkövetkezendő oldalakon egy olyan önálló tanulásra képes útkereső program részletes dokumentációja olvasható, mely egy véletlenszerűen generált, vagy a felhasználó által felépített, csapdákkal és falakkal ellátott pályán képes megtalálni az előre kitűzött célt, illetve képes felismerni, ha ez a cél nem elérhető számára. Az alkalmazásban szereplő karakter az indítást követően semmilyen kezdő ismerettel nem rendelkezik az elhelyezett csapdák létezéséről, tulajdonságairól, ezeket az információkat a pályán való közlekedés során szerzett tapasztalatai alapján szerzi meg.

A karakter környezetének érzékelését, a pályán való közlekedését, továbbá a szerzett tapasztalatok feldolgozását egy *biológiai modell* segítségével valósítottam meg. Ez a modell egy absztrakt szimulációja egy élőlény szervezetének, ahol a test különböző részei egy-egy különálló tevékenységért felelnek, ezek a testrészek pedig jelek küldésével, fogadásával kommunikálnak egy központi aggyal. Mivel egy élőlény szervezete is teljesen önálló működést folytat, ezért a szimulált változatának sincs szüksége semmilyen külső inputra, módosításra, egy új pályát kapva az indulástól a keresés végéig egyedül funkcionál. A felhasználó csak közvetetten tudja befolyásolni a modell működését. Az alkalmazás rendelkezik egy pályaszerkesztővel, mely segítségével saját akadályokat állíthatunk a program elé, a készített műveket elmentve pedig bármikor betölthetjük.

Az előbbiekben részletezett feladat megoldásához szükség volt egy olyan keretrendszerre, mely egyszerűvé, gyorssá és könnyen átláthatóvá teszi a teljesen elkülönülő objektumok közti kommunikációt. A Qt alkalmazás-keretrendszer 5-ös verziójában továbbfejlesztett *Signals & Slots* eszköztár tökéletesnek bizonyult a biológiai modell elkészítéséhez, ezért a választáson nem kellett sokat gondolkodni.

Az alkalmazás egy C++ nyelven írt, teljes mértékben objektum-orientált, *Model-View-Persistence* (Modell-Nézet-Perzisztencia) rétegekre bontott *Qt Widgets Application*, melyben a modell két további szintre rétegződik. Az MVP architektúra alól kivételt képez a pályaszerkesztő modul, mely egy különálló minialkalmazásként is felfogható, hiszen nem áll kapcsolatban a fő program magját képező modellel.

A fejlesztést megelőzően négy fontos irányelvet fogalmaztam meg magamnak, melyeket minden felmerülő probléma esetén szem előtt tartottam.

- Az első ilyen irányelv az **olvashatóság**. Minden kódot igyekeztem a lehető legkönnyebben átlátható és érthető módon írni, hiszen fontos, hogy sem egy idegen szempár ne zavarodjon össze olvasás közben, sem saját magunk dolgát ne nehezítsük meg, ha egy régebben írt kódrészlet módosítására, vagy átnézésére van szükség. Ebbe beletartozott a kód megfelelő tagolása, különböző névterek alkalmazása, operátor túlterhelések, illetve segédfüggvények definiálása. Az olvashatóság és az esetleges programozói hibák elkerülésének érdekében a programban minden lehetséges esetben *smart pointereket* használtam, melyek a memóriaszivárgás problémáját orvosolják.

- A **hatékonyság**ot hasonlóan elengedhetetlennek találtam, azonban a feladat milyenségéből adódóan ez az irányelv átértékelésre került. Természetesen a kitűzött „tanuló algoritmus” egyszerűbb és hatékonyabb módon is megoldható, mint ahogy jelenlegi program felépítése engedi, azonban a biológiai modell érdekében több helyen elcseréltem a hatékonyságot a modell pontosabb szimulálásának érdekében, ennek következtében a fő cél nem az volt, hogy a feladat hogyan oldható meg a leghatékonyabban, hanem az, hogy a biológiai modell egyes részeit hogyan tudom a lehető legjobban optimalizálni úgy, hogy közben megtartom az eredeti elképzelést. Az előbb említett cél érdekében aktívan kerestem módokat a kód gyorsítására. Minden programrész írásakor a nyelv részletes dokumentációját olvasva, tudatosan választottam a C++ szabvány könyvtáraiban megtalálható konténerek, illetve függvények közül. A fejlesztés során gyakran mértem a bizonyos programrészek sebességét, illetve hasonlítottam össze különböző implementációs megoldásokat, hogy megtalálhassam a legoptimálisabb megoldást.

- Igaz, hogy egy ehhez hasonló dolgozat esetében nem áll fenn a későbbi módosítások, újítások „veszélye”, azonban én mégis nagyon fontosnak tartottam, hogy legjobb tudásom szerint **rugalmas**sá tegyem fejlesztői szempontból a kódot. A függvényeket jól paraméterezhetően írtam meg annak érdekében, hogy egy-egy újítás esetén ne kelljen sokat módosítani a már létező programkódon. A metódusok a paraméterezhetőség mellett több mennyiségű, és több típusú adatra fel vannak készülve, mint amit a jelenlegi program biztosít, így egyszerűen vehetünk fel új fajta csapdákat, célokat, módosító effekteket. Ezek mellett bizonyos helyeken megtalálható „lehetetlen” hibák kezelése, melyek az alkalmazás mostani állapotában nem fordulhatnak elő, annak érdekében, hogy egy későbbi változás következtében egy korábban megírt függvény továbbra is jól működjön.

- Tanulmányaim során már megismertem az **objektum-orientált**ság előnyeit, ezért a projekt során végig tartottam magam a benne foglalt elvekhez. Ezen elvekből kifolyólag kerültem az osztályok közti baráti (*friend*) kapcsolatot, továbbá az osztályok sem rendelkeznek publikus adattagokkal. A globális változók használata ugyan nem sért egyetlen elvet sem, ennek ellenére ezek használatát is igyekeztem kerülni.

2. Felhasználói dokumentáció

2.1. Kitűzött feladat

A feladat egy önálló működésre képes program megvalósítása, melyben az alkalmazásban szereplő karakter egy előre meghatározott cél eléréséért küzd. A játékban szereplő pályát egy *35 x 35*-ös négyzetes mátrixként ábrázoljuk, melyeken a karakter egy lépésben a négy égtáj valamelyikének irányába tud mozogni. Vizuálisan ezt egy gombtábla segítségével jelenítjük meg, ahol a különböző színű gombok jelzik a különböző típusú mezőket.

A felhasználó az indítás előtt képes legyen befolyásolni a generálandó pálya nehézségét, ez a csapdák mennyiségében és veszélyességében nyilvánuljon meg. A felület biztosítson lehetőséget arra, hogy a futás bármely pontján szüneteltethető legyen az események folyása, továbbá a program futását lehessen lassítani és gyorsítani, a könnyebb megfigyelés vagy gyorsabb végeredmény érdekében. A felhasználónak ne kelljen megvárni, amíg befejeződik egy menet, bármikor indíthasson vagy betölthessen egy újat anélkül, hogy ez megzavarná az alkalmazást.

Fontosnak tartottam, hogy legyen elérhető egy pályaszerkesztő funkció, ahol az összes lehetséges mező közül választva tetszőlegesen lehet építkezni. Az itt elkészített pályákat el lehessen menteni fájlba bármilyen néven, majd ezeket később a felhasználó bármikor betölthesse. A pályaszerkesztő ne engedje meg a szabálytalan pályák mentését.

Fontos, hogy a minden menet véges időn belül befejeződjön, a program ismerje fel a sikert és a kudarcot is egyaránt, továbbá egyik menet se húzódjon el a kelleténél tovább. A karakter kezdetben nem ismeri a pálya teljes részét, csak a közvetlen környezetét látja, mely segítségével minden lépésben felépíti egy kis részét a teljes pályának. A karakter ne járja be a pálya azon szakaszait még egyszer, melyeket már egyszer felfedezett és képes legyen az átélt tapasztalatok alapján tanulásra, ezek alapján pedig ne tegye ki magát felesleges veszélynek a jövőben.

2.2. Telepítés és rendszerkövetelmények

Az alkalmazás a szakdolgozathoz mellékelt CD segítségével telepíthető fel egy 64 bites Windows 10 operációs rendszer alatt futó számítógépre. A program a CD-n található „BioModel” címmel ellátott mappát felmásolva egy tetszőleges helyre a számítógépen, a „biomodel.exe” végrehajtható fájl futtatásával indítható el. A sikeres telepítéshez a gép merevlemezén minimum 50MB helyet kell biztosítani. A program jó átláthatósága érdekében legalább 1600 x 900-as felbontáson ajánlott futtatni, ugyanis az alkalmazás ablakának mérete nem változtatható.

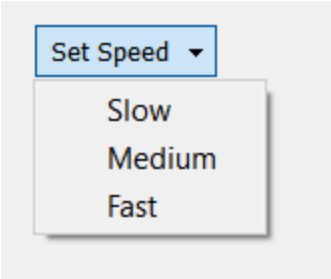
2.3. A fő program használata

A programban szereplő karakter az alkalmazás indításakor „születik meg”, és egészen addig „él”, amíg a felhasználó be nem zárja az applikációt. Ez azt jelenti, hogy minden tapasztalat, amit a karakter egy menetben megszerez, azt tovább viszi az elkövetkezendő menetekre is, akár véletlenszerűen generált, akár betöltött pályáról van szó, azaz minél több menetben vett részt, annál pontosabban tudja megítélni a veszélyeket.

Indítást követően a program egy Windows ablakban jeleníti meg a felhasználói felületet. Az ablak tetején található gombokkal lehet navigálni a különböző funkciók között, míg a felület kezdetben üres része a pályát ábrázoló rácsnak van fenntartva.

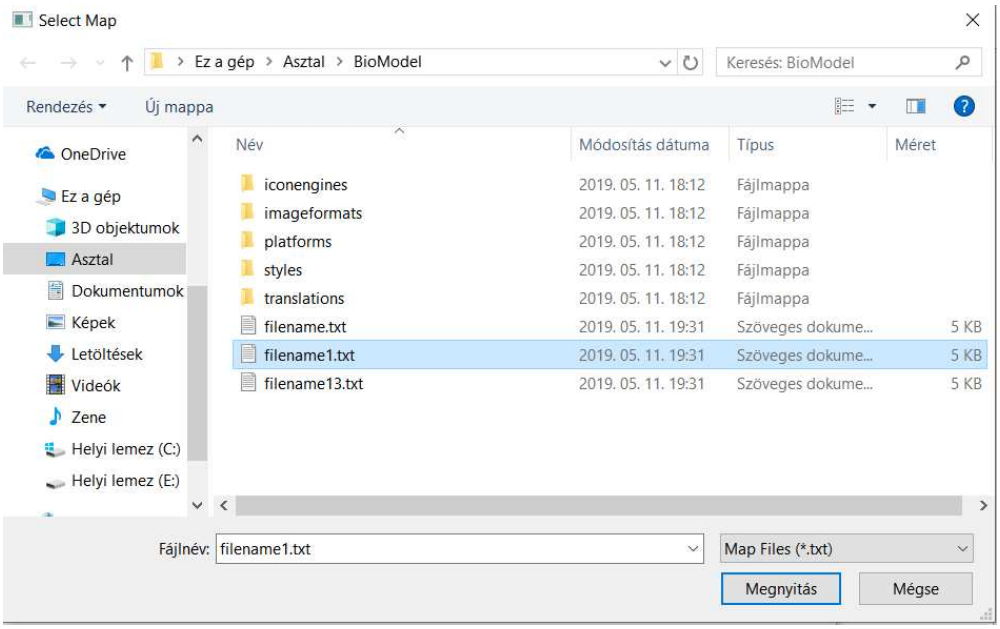
2.3.1. Felhasználói felület menet indítása előtt

- **Start Game** gomb: Megnyomásával elindít egy menetet, továbbá megjeleníti a gombrácsot, melyen figyelemmel kísérhetjük az eseményeket.
- **Easy, Normal, Hard** rádiógombok: Befolyásolják, hogy a **Start Game** gomb megnyomását követően milyen komplexitású pálya generálódik. Egyszerre csak egy jelölhető ki, alapértelmezetten **Easy**, azaz könnyű mód van beállítva.
- **Pause** gomb: Kikapcsolva. Bővebben a 2.3.2 fejezetben olvasható.
- **Set Speed** menüs gomb: Rákattintva egy legördülő menün választható ki a játék sebessége a **Slow, Medium, Fast** opciók közül. Kiválasztás nélkül az alapértelmezett érték **Medium**.

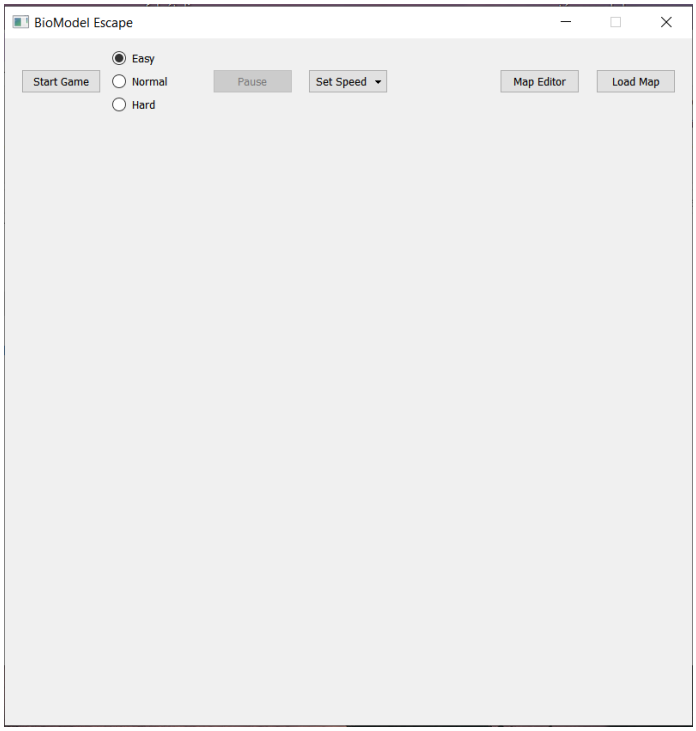


2.1. Sebességállító menü

- **Map Editor** gomb: Megnyomásával felugrik a pályaszerkesztő ablak. Bővebben a 2.4. -es fejezetben olvasható.
- **Load Game** gomb: Megnyomásával egy Windows tallózás ablak jelenik meg, „Select Map” címmel, ahol kiválaszthatjuk a számítógépre korábban elmentett pályákat.



2.2. Betöltő ablak

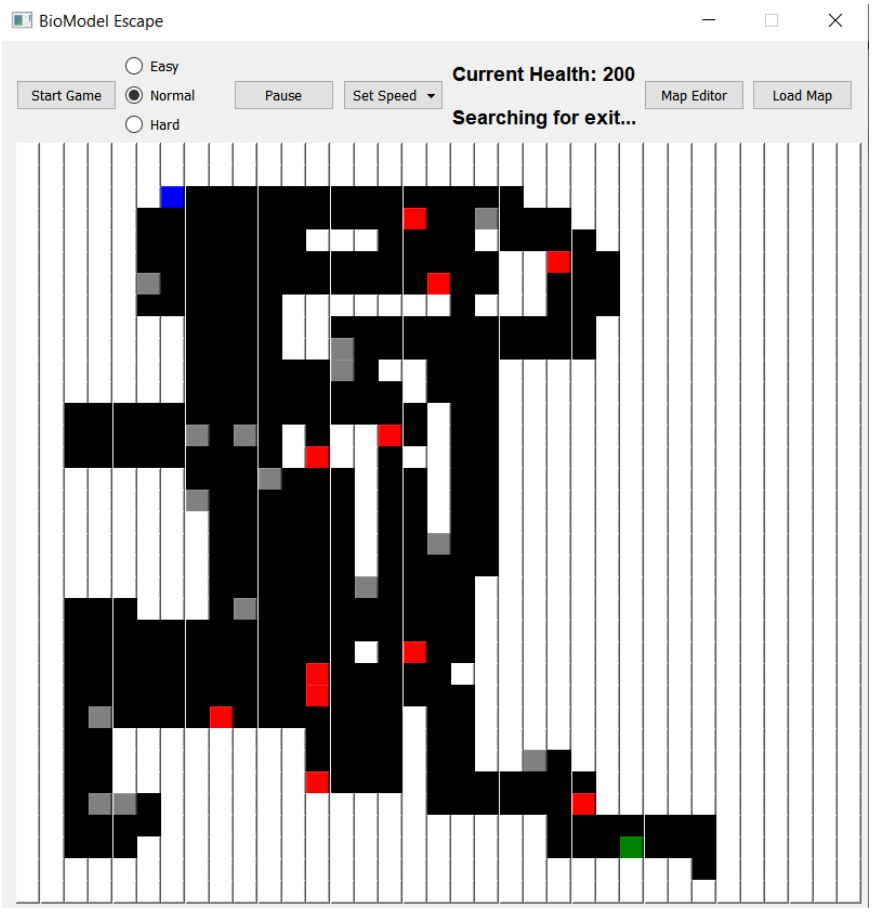


2.3. Program indítását követő ablak

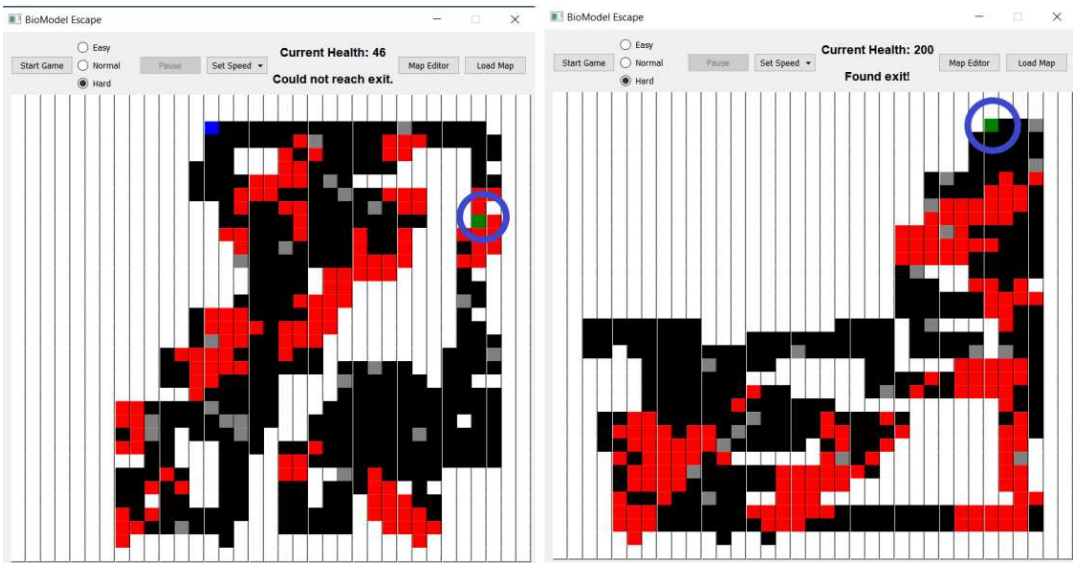
2.3.2. Felhasználói felület menet indítását követően

- **Pause** gomb: A **Start Game** lenyomásával válik elérhetővé ez a gomb, mellyel temporálisan megállíthatók a képernyőn látható események. A menet végén újra elérhetetlenné válik a gomb.
- **Élet és Információs** címke: **Start Game** megnyomását követően láthatóvá válik ez a két címke. A felső címke a karakter aktuális életét közli a felhasználóval, mely minden lépésben frissül, az alsó információs címke pedig jelzi, hogy milyen stádiumban tart a menet. Amennyiben még folyik a keresés a „Searching for exit” felirat látható, játék vége esetén pedig a végeredménytől függően „Found exit!” (ha megtalálta a karakter a kijáratot, 2.5-ös ábra) vagy „Could not reach exit.” (nem tudta elérni a kijáratot, 2.5-ös ábra) feliratok olvashatók.
- **Gombrács**: Az ablak legnagyobb részét egy *35x35*-ös gombrács foglalja el, melyen különböző színkódokkal ellátott gombok találhatók. A színekhez tartozó jelentések a következők:

- **Fehér**: Fal, átjárhatatlan és átláthatatlan terület.
- **Fekete**: Padló, bejárható terület
- **Piros**: Tűz, csapda, melyre, ha rálép a karakter az elkövetkezendő két lépésben véletlenszerű súlyosságú égési sérüléseket szenved. Újabb tűzre lépés esetén az égési sérülések időtartama plusz két lépéssel meghosszabbodik. **Normal** és **Hard** nehézségi fokozaton a tűz minden lépés elején bizonyos eséllyel terjed, minél több tűz mező van egymáshoz közel, annál nagyobb valószínűséggel terjed tovább.
- **Szürke**: Tüske, csapda, ami rálépés esetén egy bizonyos intervallumon belül véletlenszerű sebzést okoz a karakteren.
- **Zöld**: A karakter, akinek a célja, hogy megtalálja a kijáratot.
- **Kék**: A cél, kijárat. Mindig a pálya felső bal sarkában helyezkedik el.



2.4. Menet indítását követő ablak



2.5. bal: sikertelen menet vége, jobb: sikeres menet vége

2.4. A pályaszerkesztő

A fő képernyőn a **Map Editor** gomb megnyomását követően egy új ablakban megnyílik a pályaszerkesztő. Ez egy kezdetben egy olyan gombrácsot jelenít meg, melyben minden mező alapértelmezetten falra van állítva. A felhasználó az ablak tetején található színes gombok egyikére kattintva kiválaszthatja, hogy milyen mezőt szeretne lerakni, majd a gombrács tetszőleges gombjára kattintva megvalósíthatja azt. A gombrácsra való kattintás esetén mindig az utoljára megjelölt mezőfajta kerül fel rá, tehát ugyanolyan fajta mező többszöri felfestése esetén elegendő csak egyszer kiválasztani azt.

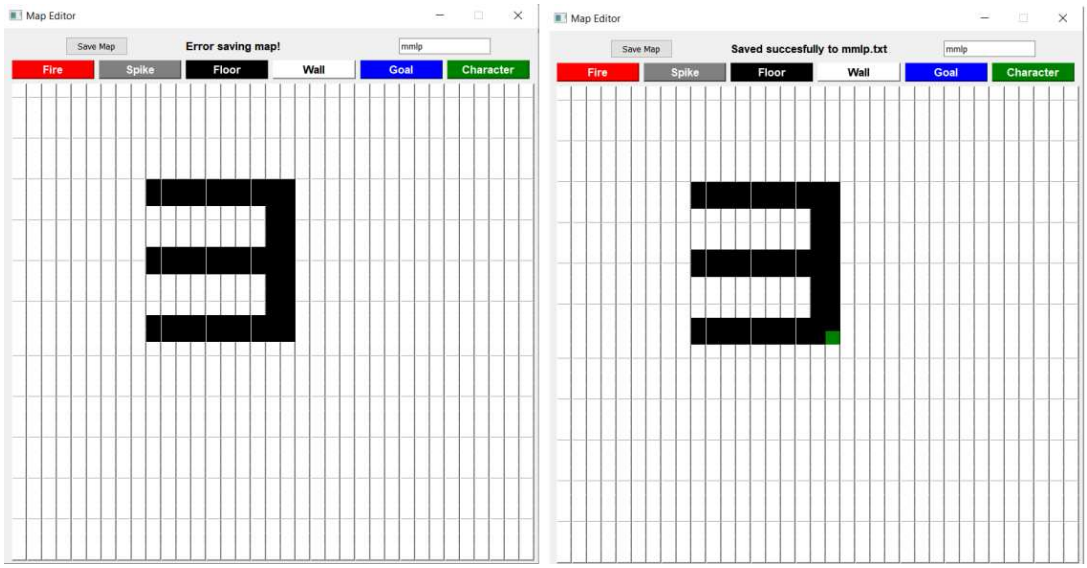
Az elkészült pályát a **Save Map** gomb megnyomásával lehet elmenteni. A mentett fájl neve az ablak jobb felső sarkában található „filename” feliratú **szövegdoboz** tartalma lesz, formátuma pedig *txt* formátumú. A mentési folyamat kezdeti lépése egy validáció, melyben a program megvizsgálja, hogy szabályos pályát épített-e a felhasználó. A szabályos pálya két feltétele:

- A pálya teljes része fal-mezőkkel van körülvéve. A szerkesztő rendeltetésszerű használata nem engedi, hogy a külső falak eltávolításra kerüljenek.
- A karakter kezdő pozíciója meg van jelölve, továbbá nem a pálya szélén helyezkedik el és bejárható mezőn található (nem fal). A szerkesztő rendeltetésszerű használata esetén az utóbbi két probléma nem fordulhat elő.

Megjegyzés: A cél mező megjelölése nem szükséges, hiszen a karakter azt is képes felismerni, ha elérhetetlen számára a kijárat.

Mentést követően az ablak tetején megjelenik egy szövegdoboz, mely közli a felhasználóval, hogy sikeres volt-e a mentési folyamat. Sikertelen mentés esetén „Error Saving” felirat látható, sikeres mentés esetén pedig kiírásra kerül a fájl teljes neve, melybe a mentés történt. A mentés mindig ugyanabba a mappába történik, ahol a futtatható állomány található, esetünkben ez a „BioModel” mappa gyökere.

A mentett fájlok később áthelyezhetők, hiszen a betöltés funkció a számítógép bármely pontjáról képes tallózni a kívánt fájlt. Ha a felhasználó befejezte a szerkesztést, akkor a jobb felső sarokban található „X” gombbal bezárhatja az ablakot, és már be is töltheti a készített pályát az alkalmazás fő ablakjából.



2.4.1. Sikertelen mentés (bal) és sikeres mentés (jobb)

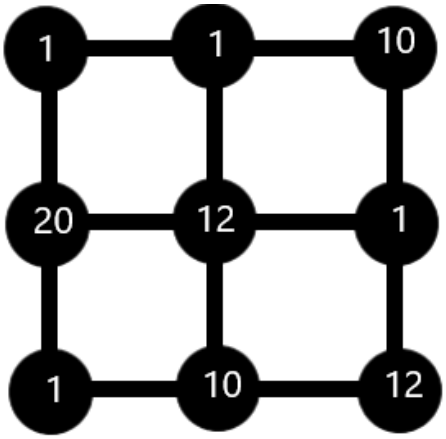
3. Fejlesztői dokumentáció

3.1. Megoldási terv

3.1.1. Fejlesztői tervezés

A 2.1.-es fejezetben specifikált feladat valójában egy jelek küldésével és fogadásával működő objektum megvalósítása, melynek minden lépésben egy tanítható gráfbejáró algoritmus játszódik le a belsejében.

A feladatban szereplő **pályát** egy négyzetes mátrixban tárolom el, de valójában úgy kell tekinteni rá, mint egy **négyzetrácsgráf**ra[1]. Ezen gráf megfelelő részgráfjain **Dijkstra algoritmus** segítségével számoljuk ki a legkedvezőbb útvonalat. Ebben az esetben nem az élekhez, hanem a csúcsokhoz szükséges súlyt rendelni, így egy *a* csúcsból *b* csúcsba vezető él költsége valójában a *b* csúcshoz rendelt súly lesz, *b* pontból *a* csúcsba pedig az *a*-hoz rendelt súly.



3.1. példa egy 3x3-as négyzetrácsgráfra, ahol csúcsokhoz rendelt súlyok találhatók

A megoldás egy fontos része a bolyongás elkerülése, vagyis semmiképpen sem előnyös, ha a játékban szereplő karakter céltalanul járja a gráfot, ennek orvoslására

[1] **Négyzetrácsgráf**: Egy n x m-es négyzetrácsgráfnak a csúcsai megfeleltethetők a sík egész koordinátájú pontjainak, ahol az x koordináták 1…n közé, az y koordináták pedig 1…m közé esnek, és két csúcs közt akkor húzódik él, hogy ha pontosan egy koordinátájukban térnek el, és ez az eltérés pontosan 1.[1]

eltároltam a bejárt részgráfot a karakter memóriájában, hogy ne felejtse el, amerre már járt, és felismerje, ha már az egész bejárható gráfot megfigyelte.

A karakter az átélt tapasztalatokat szintén elmenti a memóriájába, méghozzá olyan formában, hogy az ideális út számlálása közben az adatok könnyen konvertálhatók legyenek egy egész számra, mely egy mezőhöz rendelt súlyt jelent.

A biológiai modellt a Qt-ben található *Signals & Slots* eszköztár segítségével valósítottam meg. A modellt **Szem**re, **Test**re és **Agy**ra osztottam fel, ezek az objektumok mind különálló osztályok. Mindegyikük képes jelek küldésére *Signal*okkal, és jelek fogadására *Slot*okkal. A megfelelő küldő-fogadó kapcsolatokat maga a **BioModel** osztály fogja felépíteni a Qt-ben definiált *connect(...)* függvénnyel.

Az önálló működés érdekében szükség van még egy modell rétegre, mely a játék futását szabályozza. Ezen a rétegen elhelyeztem egy időzítő objektumot, aminek a ritmusára fog futni a program. Ez könnyen gyorsítható, lassítható, és ezzel elkerültem, hogy bármilyen külső inputra, vagy függvényhívásra szükség legyen ahhoz, hogy a biológiai modell zökkenőmentesen működhessen.

A projektet Modell-Nézet-Perzisztencia architektúrában építem fel, mely alól egyszerűség kedvéért kivételt képez a pályaszerkesztő osztály, mely egyben valósítja meg a nézetet és a modellt. Az adatelérésre a **DataAccess** névtérben elhelyezkedő függvények szolgálnak, míg a megjelenítéshez **QWidget** objektumokat használtam.

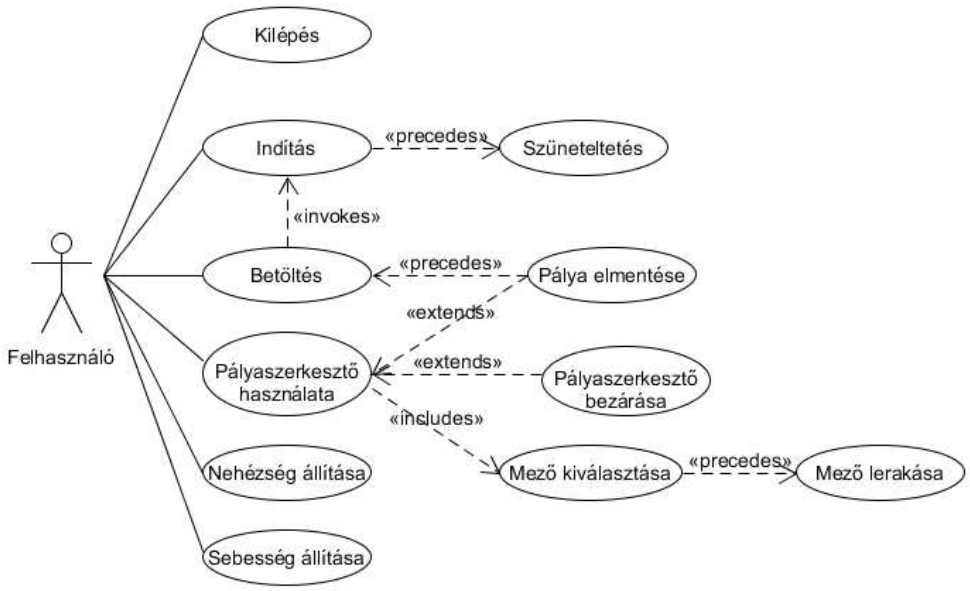
A fejlesztés során fontosnak találtam a hatékonyság mérését és a komplex megoldások egyszerű ellenőrzését. Ezek érdekében definiáltam a fejlesztést segítő makrókat, melyek segítő debug üzeneteket írnak a konzolra, kirajzolják az adott részgráfokat egy fájlba, vagy lemérik egy programrész futásának idejét. Ezek egy **Toolbox** névtérben helyezkednek el, használatukat ugyanitt ki lehet kapcsolni preprocesszor direktívákkal. Mivel az említett eszközök csak a fejlesztést segítették, és a felhasználói élményt semmilyen formában nem befolyásolják, ezért a CD-n mellékelt verzióban kikapcsolásra kerültek.

3.1.2. Felhasználói esetek

Az alkalmazásban a felhasználó az alábbi funkciókat érheti el:

- Menet indítása
- Pályanehézség állítása
- Futás szüneteltetése
- Futás sebességének állítása
- Pálya betöltése
- Pályaszerkesztő használata
- Pálya mentése
- Kilépés

Az alábbi ábrán látható az imént említett felhasználói esetek Use Case diagramja.



3.2. Felhasználói esetek diagram

3.1.3. A program belső struktúrája

A projekt **Nézet** rétegét egy **QWidget** osztályból leszármaztatott **MainWidget** nevű osztály alkotja. Ezen helyezkedik el az összes, a felhasználó számára látható információ és vezérlő. A **MainWidget** tartalmazza a pályaszerkesztőt, melyet **EditorWidget**-nek neveztem. Az **EditorWidget** objektum csak a Map Editor gomb lenyomását követően jön létre, és a **MainWidget** ablak bezárását követően is képes tovább működni.

A modell réteg felső rétegét a **GameModel** osztály reprezentálja, az alsó rétegét pedig a **BioModel** osztály. A modell réteghez tartozik ezek mellett a **DungeonMap** osztály, ami az alkalmazásban szereplő pálya reprezentációjáért felel.

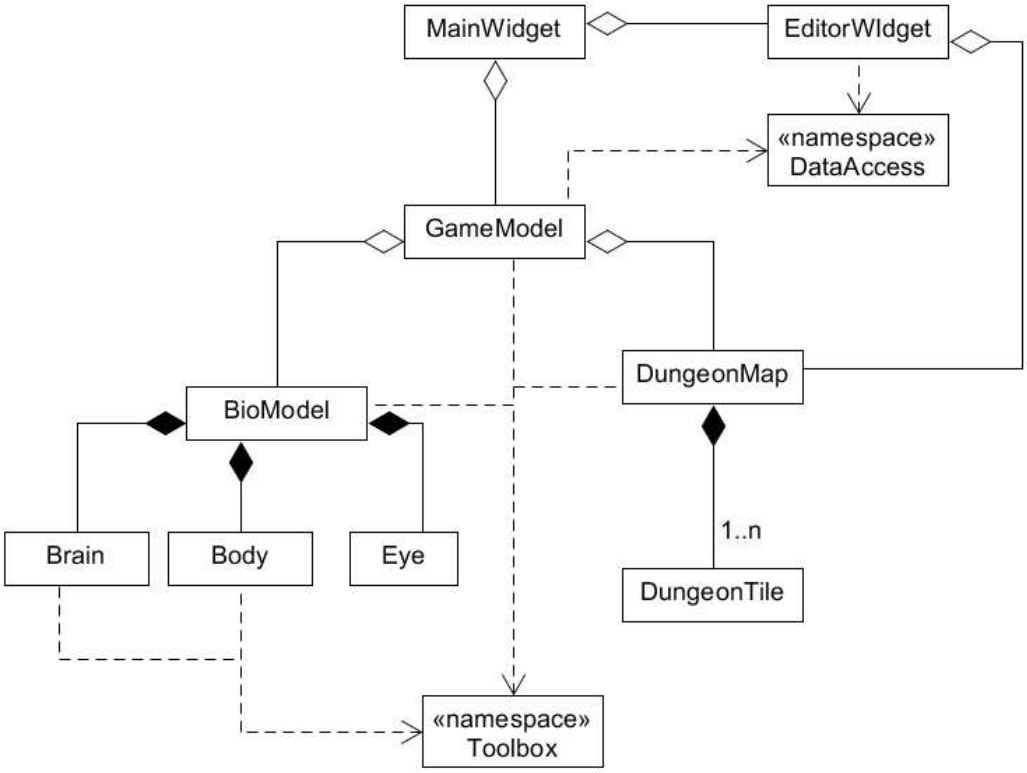
- A **GameModel** osztály felelős azért, hogy a felhasználó által adott inputok lekezelésre kerüljenek, továbbá itt helyezkedik el az időzítő objektum, ami összekapcsolva a **BioModel**-lel, a játék ritmusát szabályozza. A **GameModel** a **Nézet** réteggel szignálokon keresztül kommunikál, ezek hatására történnek változások a felhasználói felületen, továbbá közvetlen elérésben áll egy, a pályát reprezentáló **DungeonMap** típusú objektummal, melyet bármikor módosíthat.
- A **DungeonMap** osztály végzi a pályagenerálást, ennek segítségével tudjuk lekérdezni egy adott mező tulajdonságait, továbbá a karakter kezdő pozícióját. Ezzel az osztállyal szoros kapcsolatban áll a **DungeonTile** osztály, mely egy adott mezőről tartalmaz minden információt, lényegében a **DungeonMap** alkotóeleme.
- A **BioModel** osztály szimulálja a biológiai modellt, ez reprezentálja magát a karaktert a játékban. Az alkalmazás indításakor rákötöm a **GameModel** időzítőjére, és ezen időzítő által kiadott szignálokat követően lép működésbe. Felel azért, hogy az őt alkotó **Eye, Body, Brain** (Szem, Test, Agy) objektumokat összekapcsolja, továbbá ellátja a **GameModel**-t szükséges információkkal.

A **Perzisztencia** réteg a **DataAccess** névtérből áll, mely megvalósítja a fájlkezeléssel kapcsolatos műveleteket. Két metódusa a mentés és a betöltés.

A fent említett rétegek mellett létezik egy **Toolbox** névtér, mely a többi réteg által használt, de nem szorosan oda kapcsolódó „utility” függvényeket, procedúrákat és

objektumokat tartalmazó könyvtár. Emellett definiáltam továbbá a fejlesztés és tesztelés elősegítésére készült makrókat is, ezeket pedig ugyanitt lehet kikapcsolni egy sor módosításával a kódban.

A főbb komponensek közti kapcsolatok az alábbi ábrán láthatók:



3.3. Fő komponensek osztálydiagramja

Az elkövetkezendő fejezetekben az imént említett komponensek részletes szerkezetét, logikáját, illetve implementációs részleteit fogom tárgyalni.

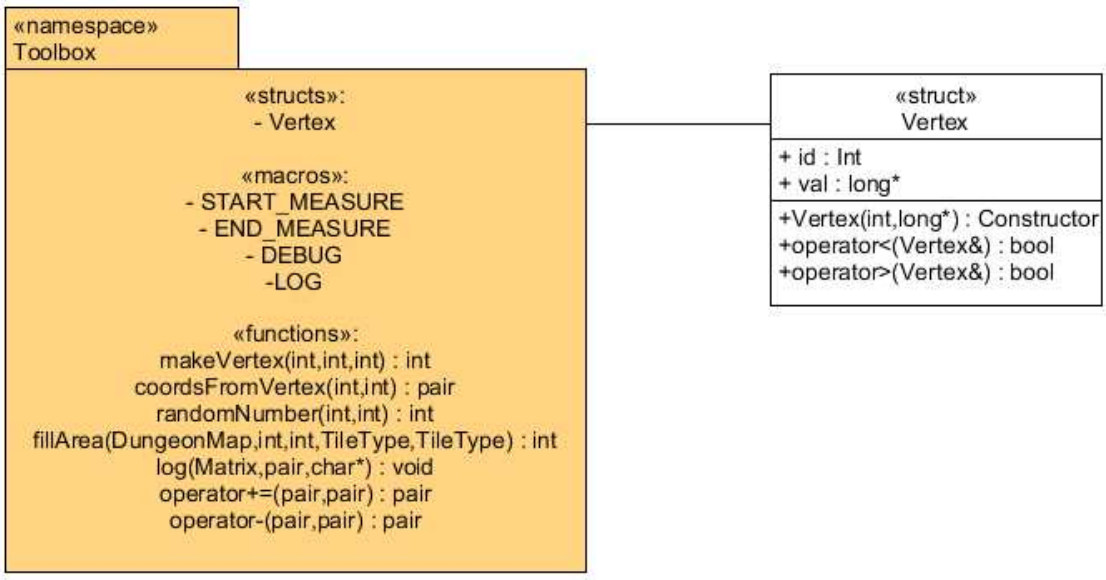
3.2. A Toolbox névtér

3.2.1. Felépítés

A **Toolbox** header fájl a fejlesztést és tesztelést segítő

- **makrók**ból
- **függvények**ből és **procedúrák**ból
- **adatszerkezetek**ből

áll. A fájlban található **makró**k nem a tényleges névtér részei, ezeket csak moduláris okból definiáltam itt. Az alábbi ábrán a fájl részletes felépítése látható:



3.4. A Toolbox névtér felépítése.
Magyarázat:
Matrix = vector<vector<DungeonTile>>
pair =két egész számból álló pár

3.2.2. Vertex, azaz a gráf csúcs adatszerkezet

A későbbiekben tárgyalt **Dijkstra algoritmus** implementációjához szükség volt egy olyan adatstruktúrára, amelyen könnyen ábrázolhatok egy olyan gráfcsúcsot, melyben a forráscsúcstól vett távolság értéke dinamikusan változik egy külső hatás által (bővebben:

TODO: fejezetszám, cím). Fontos, hogy ezeket a csúcsokat be tudjuk azonosítani, erre szolgál az *id* mező, ami egy mátrix koordinátára könnyen visszaalakítható egész szám.

A Vertex konstruálásának az egyetlen módja, ha megadjuk mind az *id*, mind a *val* paramétereket.

A ***kisebb*** és ***nagyobb*** (***operator<*** *és* ***operator>)*** operátorok túlterhelésére azért volt szükség, hogy könnyen meg tudjuk állapítani, hogy két csúcs közül melyiknek kisebb vagy nagyobb az aktuális távolsága a forrás csúcstól. Ezek az operátorok a *val* mező összehasonlítását végzik. Mivel ez az adattag egy pointerként van implementálva, melyeket más folyamatok megváltoztathatnak, ezért fel kell készülni hibás használatra, és az érvénytelen pointerek esetét is helyesen le kell kezelni. A ***business logic*** elvét alkalmazva, miszerint nem előnyös ismétlődő kódot írni, a *nagyobb* operátor vissza van vezetve a *kisebb* operátorra.

3.2.3. Makrók

A fejlesztés során minden sebesség szempontjából kritikus programrész futási idejét lemértem, és több fajta implementáció kipróbálásával az mellett döntöttem, amelyik a leggyorsabb volt (bővebben: *TODO: fejezetcím, szám)*. Erre a mérésre használt makrók a ***START_MEASURE****(name),* ***END_MEASURE****(name)*. A makrók párban használatosak, a mérés az előbbivel kezdődik el, majd az ugyanazzal a névvel ellátott *END_MEASURE* makróval záródik le. Ekkor a névvel ellátott, mikroszekundum pontosságú eredmény a *qInfo()* függvény segítségével kiírom fejlesztői környezetben az *Application Output* ablakba. Ezek a makrók *std::steady_clock* objektumok segítségével számolják ki a két időpontot, majd az *END_MEASURE* egy *std::duration* objektummal határozza meg a két időpont közt eltelt időt. A profilozás pontosságának érdekében esett a választásom a makrókra, hiszen azoknak a futási idejű költsége minimális.

A ***DEBUG****(message)* makró segítségével egy tetszőleges üzenetet lehet kiíratni a fejlesztői környezetben található *Application Output* ablakba. A *message* üzenet mellett kiírom a függvény nevét (__FUNCTION__), illetve a „[DEBUG]” szót a könnyebb eligazodás kedvéért.

A tesztelés során szükséges volt ellenőrizni, hogy a karakter valóban jó irányba halad-e, továbbá tényleg nem juthat el a kijáratig megállást követően. E célból írtam a ***LOG****(map,pos,fname)* makrót, mely a beadott DungeonMap, *std::pair<int,int>* és *const*

*char** típusú paraméterekkel meghívja a **log**(…) függvényt, melyet a következő alfejezetben ismertetek.

Természetesen az előbb említett eszközök csak a fejlesztői szempontból relevánsak, nem ideális, ha a felhasználó gépét „szemeteljük” log fájlokkal, vagy lassítjuk a program futását felesleges számításokkal, ezért ezeket a makrókat csak a ***MONITOR_MODE*** definiálásával lehet bekapcsolni.

3.2.4. Utility függvények

A névtérben található ***makeVertex****(i,j,matrixSize)* és ***coordsFromVertex****(a,matrixSize)* függvények egy négyzetes mátrixban található mező koordinátáit alakítják egy egyedi azonosítóvá, illetve vissza. Mindkét függvénynek szüksége van utolsó paraméterében egy *size_t* típusú bemenetre, melyben megkapják a mátrix méretét, ahonnan származik a koordináta vagy azonosító. Az előbbi függvény első két paramétere az átalakítani kívánt nemnegatív koordináták, az utóbbi első paramétere pedig a visszaalakítani kívánt nemnegatív szám. A gyakorlatban ezekkel a függvényekkel készítek egy Vertex objektumnak *id*-t, illetve alakítom vissza az *id*-t egy mátrixbéli koordinátává.

A projektben az egyik leggyakoribban használt eszköz a ***randomNumber****(min,max)* véletlenszám-generátor. Bemenetében meg kell adni a *min* és *max* paramétereket, mely egy zárt intervallum alsó és felső pontjaként van értelmezve. Mivel a függvény a C-ben található *Standard Library rand()* függvényét használja, ezért minden hívás elején biztosítanunk kell azt, hogy az *srand()* metódus már meg legyen hívva egyszer, és ez a hívás program futásonként pontosan egyszer történjen meg. Ennek érdekében egy statikus *bool* változó fogja jelezni, hogy már megtörtént-e a seedelés. Amennyiben ez a változó hamis, a jelenlegi rendszeridővel (a <time.h> könyvtárban található *time(nullptr)* függvényhívás segítségével) fogom meghívni az *srand* függvényt, majd igazra állítom az említett boolean változót.

A ***fillArea****(map,x,y,toFill,filler)* függvényt csupán a teszt projektben (bővebben: *TODO: fejezetcím, szám*) használtam. Célja az, hogy ellenőrizze, hogy a generált pálya minden nem fal típusú pontja bejárható-e egy adott pozícióból, de jól paraméterezhetőségének köszönhetően bármilyen összefüggő terület feltölthető bármilyen típusú mezővel. A metódus a bemenetben egy *DungeonMap* típusú input változót (referencia) vár, melyet a futás során fel fog tölteni. Az *x* és *y* paraméterek jelzik

azt a kezdő koordinátát, ahonnan a töltés kezdődik. A *toFill* paraméterben adható meg, hogy milyen típusú mezőt írjon felül a függvény *filler* típusú mezővel a beadott *DungeonMap*-en.

A *fillArea* egy rekurzív függvény, mely a mátrix adott pontjáról elindulva a négy szomszédos koordinátájú pontjára terjed. Ha a vizsgált mező megegyezik a *toFill* paraméterben megadottal, akkor 0-val tér vissza, egyébként folytatódik a rekurzív hívás a szomszédokra. A függvény visszatérési értéke az az egész szám, ahány mezőt lehetett feltölteni egy *toFill* típusú mezők által határolt területen.

Az ***operator+=*** és ***operator-*** függvények a C++ <utility> könyvtárában található *std::pair<int,int>* objektumnak valósítják meg az inkrementálás és kivonás műveleteket. Az előbbi egy *pair<int,int>&* referenciával tér vissza a művelet egymásba ágyazhatóságának kedvéért, míg az utóbbi érték szerint adja vissza az eredményt. A műveleteket a matematikában lévő koordinátarendszer pontjai közti műveletekhez hasonlóan valósítottam meg.

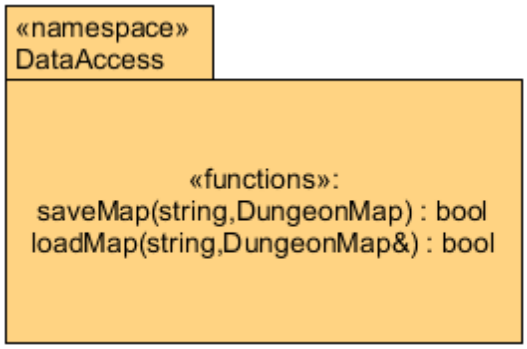
A ***log****(m,pos,fname)* függvénynek két példánya létezik, a különbség az *m* paraméter típusában van. Az első változatban egy *std::vector<std::vector<DungeonTile*>>* konstans referenciát fogad, a másikban pedig egy *std::vector<std::vector<DungeonTile>>* típusú konstans referenciát. Erre azért volt szükség, mert a programban a memória megtakarítás és hatékonyság szempontjából nem mindenhol tárolok teljes másolatokat a mátrixokról, hanem ahol ez lehetséges, másolás helyett a másik mátrix elemeire mutató pointereket mentek le. Ezt a konstrukciót használva mindkét típus esetén egyszerűen meghívható a függvény. A további két paraméter egy egész számokból álló koordináta, illetve egy *const char**, ami a fájl nevét tartalmazza.

A *log* függvényben a C++ <fstream> headerjében definiált *std::ofstream* objektum segítségével írom ki a megfelelő információkat (mátrix felépítése, karakter pozíciója) az *fname* paraméterben átadott fájlba. A program futása közben a fájl tartalmát nyomon követve könnyen ellenőrizhető és megfigyelhető a játékban szereplő karakter által letárolt részmátrixok, és az azokon elfoglalt relatív pozíciója.

3.3. A DataAccess névtér

3.3.1. Felépítés

A **DataAcccess** névtér biztosítja az adatelérést, azaz a **Perzisztencia réteget** a projektben. Két függvényt definiáltam benne, ezek a ***saveMap*** függvény és a ***loadMap*** függvény. A névtér felépítése az alábbi ábrán látható:



3.5. A DataAccess részletes felépítése

3.3.2. Mentés

A ***saveMap****(fname,mapData)* metódus lehetőséget biztosít arra, hogy egy paraméterként megadott pályát (*DungeonMap* objektumot) egy tetszőleges fájlba (*fname* paraméter) elmentsünk. A bemenő paraméterek mind konstans referenciaként adom át, hiszen a függvény csak olvas az inputokból.

Annak ellenére, hogy az alkalmazás rendeltetésszerű használata során csak egy fajta szabálytalan pálya érkezhet (nincs megjelölve kezdő pozíció a karakternek), ennél jóval több esetet vizsgáltam, hiszen egy későbbi változás, vagy esetleg a függvény egy másik projektbe való átvitelének következtében felmerülhetnek más problémák is, mint amik jelenleg jelentkezhetnek. Ennek szellemében a függvény az alábbi hibakezeléseket végzi:

- A *mapData* objektumban lévő mátrix mind a négy oldalán csak Fal típusú mezők találhatók.
- A karakter kezdő pozíciója meg van jelölve és nem a pálya valamely szélső mezőjén helyezkedik el.
- A karakter kezdő pozíciója bejárható területen (nem fal) található.

Amennyiben a fenti feltételek közül valamelyik nem teljesül, a függvény *hamis* értékkel tér vissza, ellenkező esetben pedig folytatódik a mentési folyamat. A fájlba írást egy már korábban ismertetett *std::ofstream* objektum segítségével valósítottam meg. Az output fájlt két részre lehet bontani: **header** és **mátrix**.

- A fájl első sora tekinthető a **header**-nek. Itt a következő információk találhatók, szóközzel elválasztva, megfelelő sorrendben: **pályaméret**, kezdő pozíció **első koordinátája**, kezdő pozíció **második koordinátája**.
- A fájl további része a **mátrix**ot reprezentáló rész. Itt szóközzel elválasztva található a *mapData* pálya összes mezőjéről való információ az objektumban fellelhető sorrendben. A mátrix sorai sortöréssel vannak elválasztva. Egy mezőt két darab információval ábrázoltam: ***TileType*** és ***TrapType***, azaz mező típus és csapda típus, melyek enumerációs osztályként vannak definiálva (bővebben: *TODO: fejezetcím, szám*). Definíciójukból adódóan a *static_cast* művelet segítségével könnyedén értelmezhetők egész számként, ezért a fájlban egy mezőt két szóközzel elválaszott egész számmal ábrázoltam T*ileType* – *TrapType* sorrendben.

Az olvasás befejeztével a függvény bezárja a megnyitott fájlt, és *igaz* értékkel tér vissza.

3.3.3. Betöltés

A felhasználó által készített pályák fájlból való kiolvasásához a ***loadMap****(fname,mapData)* függvényt definiáltam. Szerepe az, hogy az *fname*-ben foglalt abszolút fájlelérési útban található fájlból kiolvassa az adatokat, majd a *mapData* output paraméterbe betöltse azokat. Ebben a függvényben már nem vizsgáltam a *saveMap*-ben ismertetett hibákat, feltételeztem, hogy csak olyan pályát szeretne betölteni a felhasználó, amit a program pályaszerkesztőjével hozott létre. Ezen ok miatt a betöltési folyamat elkezdése előtt a két dolog, amit vizsgáltam az az, hogy a paraméterként megkapott fájl megnyitható-e, illetve, hogy a fájl **header** részében található szükséges információk mindegyike jelen van-e.

Ezt követően beolvastam a **header**-t, mely után megvizsgáltam, hogy az itt található *size* paraméter megegyezik-e a bemeneti *mapData* méretével. Az alkalmazás jelenlegi formájában ez a vizsgálat mindig igaz lesz, azonban a már korábban ismertetett

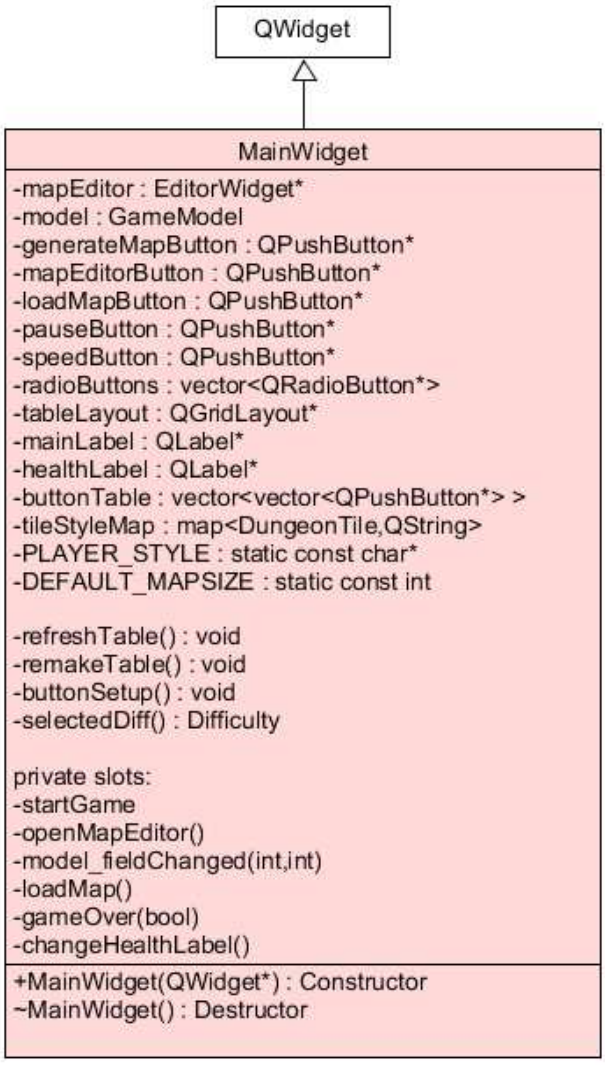
irányelvek okán nem vettem ki. A **header** beolvasása után egy ciklusban kettesével olvasva a fájl **mátrix** részét a *mapData* adott indexű elemének mező típusát és csapda típusát az első olvasott egész szám *TileType*-ra és a második olvasott egész szám *TrapType*-ra *static_cast*-olt alakjára változtattam.

Olvasást követően a függvény bezárja a megnyitott fájlt, majd *igaz* értékkel tér vissza.

3.4. A nézet réteg (MainWidget osztály)

3.4.1. Felépítés

A projekt nézet rétegét a **QWidget**-ből leszármaztatott **MainWidget** osztállyal valósítom meg, mely felelős a felhasználói felület megjelenítéséért és a **modell** réteggel való összekapcsolásáért. A MainWidget osztály felépítése az alábbi ábrán látható:



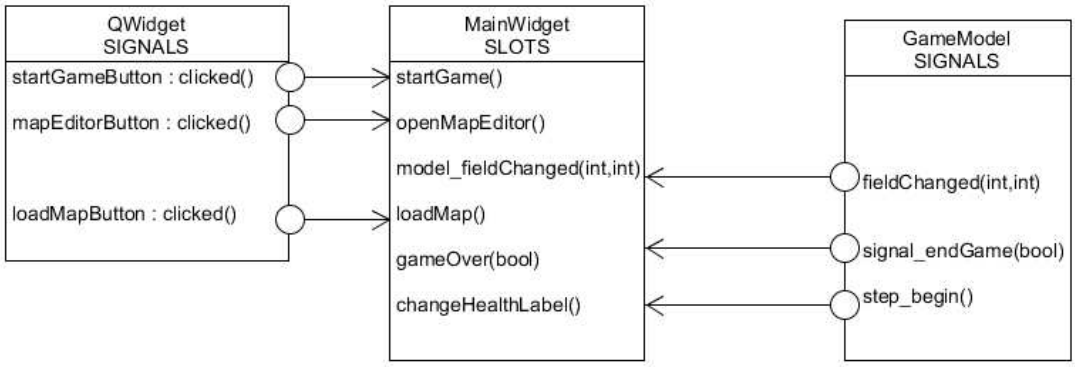
3.6. MainWidget osztály részletes szerkezete

Megjegyzés: A képen látható adattagok típusát követő * jelölés *smart pointert* jelent, a „private slots” pedig azon privát függvényeket, melyek képesek egy másik objektumtól kapott jel fogadására.

3.4.2. Szignál kapcsolatok

A **Nézet** réteg és a **Modell** réteg közötti szignál kapcsolatok nagyrészt egyoldalúak, hiszen a **MainWidget** közvetlenül eléri a **GameModel** osztályt, ezért legtöbb esetben nincs szükség az ilyen irányú szignálok alkalmazására. Ez alól kivételt képez a *PauseGameButton clicked()* eseménye, melyről bővebb információ a *TODO:fejezet* ben olvasható. A fordított irány ezzel ellentétben nagyon fontos, hiszen a nézet ezek alapján kap értesítést arról, hogy a modellben történt egy olyan változás, amit a felhasználói felületen is ábrázolni kell.

A **MainWidget** által tartalmazott *QPushButton* objektumok *clicked()* szignáljait is szükséges fogadni, hiszen így ez alapján lehet a felhasználói inputot lekezelni. A következő ábra bemutatja az osztály szignál kapcsolatait:



3.7. MainWidget osztály szignál kapcsolatai

3.4.3. Az osztály konstruktora

A **MainWidget** osztály a <QWidget> Qt könyvtárban definiált **QWidget** osztály leszármazottja. A program *main()* függvényében a **MainWidget** példányosítását követően a QWidget osztályban definiált *show()* tagfüggvény meghívására jelenik meg a felhasználói felület. Annak érdekében, hogy használhassam a *Signals & Slots* eszköztárt, az osztály privát részében szükség volt a **Q_OBJECT** makró beillesztésére. Az itt deklarált *slot*-ok privát láthatóságúak, hiszen a **MainWidget** osztályon kívül nincs más objektum, ami ezekre szignálokat küldene.

A felhasználói felületnek szolgáló ablakot az objektum **konstruktorában** építem fel. A **konstruktor** paraméterként egy QWidget pointer típusú *parent* paramétert vár, alapértelmezetten *nullptr* értékkel. Kezdetben meghívódik a **QWidget** szülő osztály konstruktora az imént említett pointerrel, ezt követően pedig inicializálásra kerül a ***_model*** egy alapértelmezett konstruktorral.

A **_tileStyleMap** változóban tároltam a különböző *DungeonTile*-ökhöz rendelt **stíluslap**okat, melyeket szintén a konstruktorban töltök fel. Ezek a stíluslapok szabják meg, hogy a felhasználó milyen színű gombokat lát a pályát ábrázoló felületen.

Az ablak tetején látható gombokat is itt hoztam létre, különös tekintettel arra, hogy a ***_pauseGameButton*** gomb kezdetben kikapcsolt állapotban jelenjen meg. A gombok eseményeit a 3.7.-es ábrán látható módon rákötöttem a megfelelő *slot*okra.

Egyetlen speciális gomb található az ablakon, ez a ***_speedButton***, mely egy **QMenu** objektummal van felruházva. A menü egyes menüpontjainak aktiválási (*triggered())* eseményét az egyszerűség kedvéért lambda függvényekre kötöttem, melyek megváltoztatják a **_model** aktuális sebességét. Sajnos a **QMenu**-vel felruházott gombok nem fogják birtokolni a hozzájuk rendelt **QMenu** pointert, *smart pointer*ekre pedig nincs felkészülve a hozzárendelést végző függvény, ezért a **destruktorban** külön törölni kell egy *delete* paranccsal a gombhoz tartozó menüt.

Az ablak elrendezését is a konstruktorban építettem fel. Az ablak globális elrendezése egy *QGridLayout* (azaz rács elrendezés), melyre a konstruktorban dinamikusan létrehozott különböző *QVBoxLayout* és *QHBoxLayout* elrendezéseket pakoltam fel. *QLayout* objektumok esetén más objektum hozzáadásával a szülő objektum birtokába veszi a gyerekeit, azaz az így hozzáadott dinamikus tárhelyen lévő elrendezéseket **nem szabad külön felszabadítani** a **destruktor**ban.

Az osztályban található két *statikus konstans kifejezés* (*static constexpr)*, amik olyan konstansok, melyeknek az értéke már fordítási időben nyilvánvaló:

- **PLAYER_STYLE**: A játékos által elfoglalt mezőhöz rendelt stíluslap
- **DEFAULT_MAPSIZE**: A pálya méretének alapértelmezett értéke. Mivel a projekt alsóbb szintjein elméletben lehetséges lenne a pályaméret változtatása, ezért ezen a szinten ezt a könnyen módosítható paramétert adom meg a modell rétegnek.

3.4.4. Jelfogadó függvények (Slot-ok)

A 3.7.-es ábrán már ismertettem, hogy melyik slot melyik szignál hatására aktiválódik, azt, hogy mi történik az egyes jelfogadó függvények hívását követően, ebben a fejezetben fogom részletezni.

- ***startGame():*** Aktiválja a játék szüneteltetését biztosító *_pauseButton*, továbbá frissíti a *_mainLabel* tartalmát. Ezt követően új játék kezdődik a **_model**-ben a DEFAULT_MAPSIZE és a rádiógombok közül kiválasztott **nehézség** alapján. Ezután egy elágazásban eldől, hogy újra fel kell-e építeni a **_buttonTable**-t (***remakeTable()*** függvény), vagy elég a meglévő gombokat frissíteni (***refreshTable()*** függvény). A gombtábla felépítése a **modell**ben található ***DungeonMap*** alapján történik, azaz a gombtábla minden pontja tükrözi a modellben levő pálya minden pontját. A pálya adott pontjának vizuális reprezentációja a gombtábla azonos pontján levő, **_tileStyleMap** által meghatározott stílusú gomb.

- ***openMapEditor():*** Létrehoz egy **EditorWidget** objektumot, majd meghívja a *show()* metódusát, ezzel megjelenítve a pályaszerkesztőt.

- ***model_fieldChanged(x,y):*** A modelltől kapott koordinátát felhasználva frissíti a **_buttonTable** (x,y) pozíciójában lévő gomb stíluslapját a **_model** (x,y) pozícióján elfoglalt mező alapján.

- ***loadMap():*** Megnyitja a fájl tallózási párbeszédablakot. Ezt követően a felhasználó által kiválasztott fájl elérési útvonalával meghívja a **modell** *loadMap* függvényét. Sikertelen betöltés esetén egy információs párbeszédablak ugrik fel, sikeres betöltés esetén frissíti a **modell** nehézségét a kiválasztott rádiógomb alapján, aktiválja a szünet gombot, frissíti a gombtáblát, illetve a címkéket.

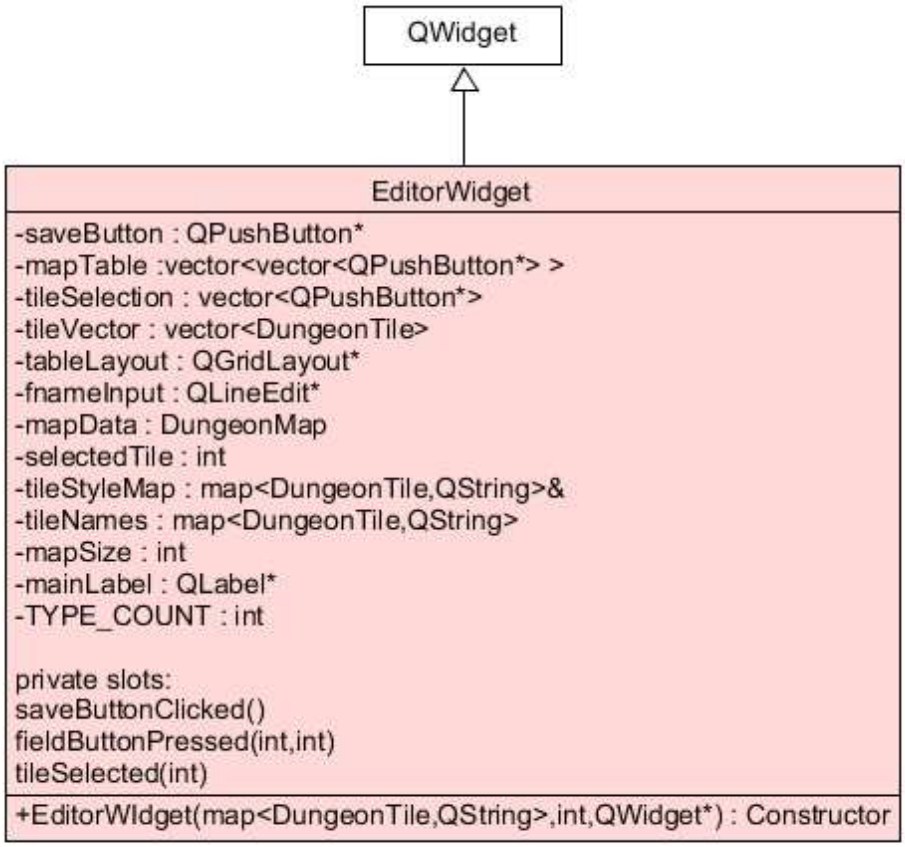
- ***gameOver(won):*** Játék végét jelző esemény. Deaktiválja a szünet gombot, majd frissíti a címkéket a *won* boolean alapján.

- ***changeHealthLabel():*** Lekérdezi a **modell**től a karakter aktuális életét, majd frissíti a **_healthLabel** címkét.

3.5. A pályaszerkesztő (EditorWidget)

3.5.1. Felépítés

A **QWidget**ből leszármaztatott **EditorWidget** osztály egy személyben felelős a pályaszerkesztő megjelenítéséért és működtetéséért. A projekt ezen részén az egyszerűség és tömörség kedvéért szándékosan mellőztem a Model-View architektúra használatát. Az osztály felépítése az alábbi ábrán látható:



3.8. EditorWidget osztály felépítése

Megjegyzés: Hasonlóan a **MainWidget** osztályhoz, a privát adattagok típusait követő * jelek itt is *smart pointer*eket jelölnek.

3.5.2. A konstruktor

A **MainWidget** osztályhoz hasonlóan ebben az osztályban is a konstruktorban építem fel a felhasználói felületet, továbbá inicializálom a működéshez szükséges változókat.

A konstruktor a korábban látott *QWidget* pointer mellett két másik paramétert is fogad: *tileStyleMap* és *size.* Az előbbi a **MainWidget**ben ismertetett *DungeonTile* objektumhoz *stylesheet*-et rendelő map objektum. Ezt a paramétert hatékonyság szempontjából nem másoltam le, hanem az inputra mutató referenciát az osztályban található *_tileStyleMap konstans referencia* objektumban tároltam el. Ezt azért tehettem meg, mert ez a paraméter mindig **MainWidget** osztályból származik. A *size* paraméter szabályozza, hogy mekkora gombrács készüljön.

Az osztályban található *_tileSelection* összhangban áll a *_tileVector* objektummal, ezek ketten biztosítják a felhasználó számára, hogy a megfelelő mezőt jelző gombra kattintva a megfelelő *DungeonTile* objektum legyen kiválasztva. Ehhez kapcsolódik a *_selectedTile* egész szám, ami a *_tileSelection* vektor (és ezzel együtt a *_tileVector*, hiszen egymást tükrözik) annak az indexét tárolja, melyre a felhasználó utoljára kattintott.

A *_tileNames* egy adott mezőhöz rendel stringet, segítségével tudom névvel felruházni a felhasználói felületen megjelenő gombokat.

A *_typeCount* konstans kifejezés a különböző lehetséges mezők számát jelzi.

A konstruktorban az alábbi szignál-slot kapcsolatokat állítom fel:

- **_saveButton : clicked() → *saveButtonClicked()***
- **_tileSelection i-edik gombja : clicked() → lambda** függvényt definiáltam, mely meghívja a ***tileSelected(int)*** függvényt az **i** paraméterrel, ugyanis a **clicked**() esemény nem ad át semmilyen paramétert, nekem azonban szükségem volt arra, hogy beazonosítsam melyik gomb lett lenyomva.
- **_mapTable[i][j] gombja : clicked()** → az utóbbihoz hasonló okból kifolyólag **lambda** függvényt definiáltam, mely meghívja a ***fieldButtonPressed(int,int)*** függvényt i és j paraméterrel.

3.5.3. A jelfogadó függvények

Az osztály három *slot*ja közül valójában csak az egyik fogad közvetlenül jelet, ez nem más, mint a ***saveButtonCicked()*** függvény. Ebben a metódusban kiolvasom az *_fnameInput* input sorban található szöveget, amit a felhasználó írt be, majd ezzel a fájlnévvel és az aktuális *_mapData* objektummal meghívom a **DataAccess** névtér *saveMap(..)* függvényét. Sikertelen mentés esetén egy hibaüzenetet írok ki a felhasználói felületre a *_mainLabel* adattag segítségével.

A másik két *slot*ot csak formalitásból definiáltam annak, valójában egy sima void függvény is elég lenne, hiszen, ahogy az előző fejezetben említettem, a tényleges szignálok lambda függvényekre vannak kötve.

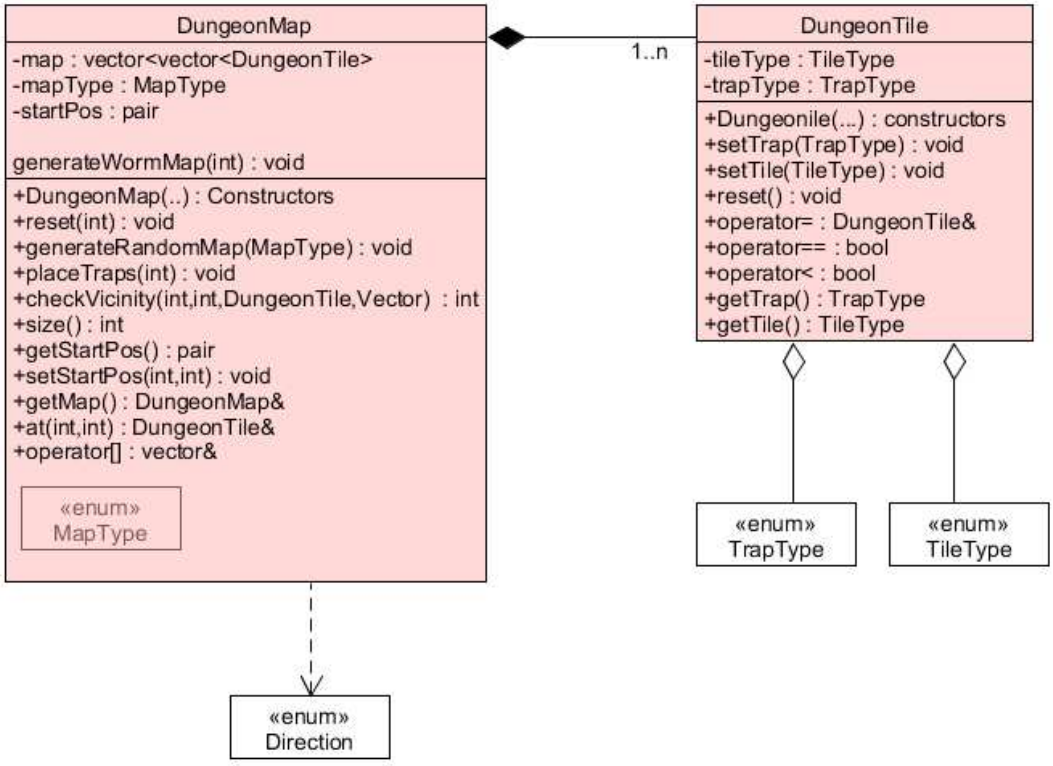
A szóban forgó függvények közül az első a ***fieldButtonPressed(x,y)***, ami abban az esetben hívódik meg, ha a felhasználó rákattintott a gombrács valamely gombjára. A lenyomott gomb koordinátái lesznek az *x* és *y* paraméterek. Ekkor kezdeti hibaellenőrzéseket követően a függvény módosítja a *_mapData* változó (x,y) koordinátán található elemét a felhasználó által korábban kiválasztott mezőre (vagyis a *_tileVector[_selectedTile]* helyen szereplő *DungeonTile* objektumra).

A ***tileSelected(i)*** eseményt az váltja ki, ha a felhasználó a mezőválasztó gombok közül megnyom egyet. A megnyomott gomb az **i**-edik indexen helyezkedik el. Hibaellenőrzést követően a *_selectedTile* adattag felveszi az *i* értékét.

3.7. A pálya reprezentációja (DungeonMap)

3.7.1. Felépítés

A programban szereplő pálya háttérben való reprezentációját a *dungeonmap.h* headerben található két osztály, a **DungeonMap** és **DungeonTile** végzi. Különös figyelmet fordítottam arra, hogy az itt megvalósított struktúrák a Qt rendszertől függetlenül működjenek. A felépítés a következő ábrán látható:



3.9. dungeonmap.h osztályainakl felépítése

3.7.2. Enumerációs osztályok

A fájlban az alábbi publikus enumerációs osztályokat határoztam meg:

- **enum class TileType** : {Floor, Wall, Unknown}
- **enum class TrapType** : {Fire, Spike, None, Goal}
- **enum Direction** : {North, East, South, West}
- **enum MapType** : {Worm, Custom}

3.7.2. DungeonTile osztály

A **DungeonTile** osztály reprezentálja a pálya egy mezőjét. Két adattagja a *_tileType* és *_trapType* határozzák meg, hogy milyen típusú mezőről van szó, illetve, hogy milyen csapda helyezkedik el rajta. Ezen adattag eléréséért **gettereket** és **settereket** definiáltam az osztályhoz.

Mivel ez az adatstruktúra a projekt szinte minden részében szerepet kap, ezért szükségesnek gondoltam, hogy a legtöbb szituációban egyszerűen és gyorsan használható legyen. Ezen okból kifolyólag **4 konstruktor**t készítettem el:

- **Default** konstruktor: A C++ által alapértelmezett konstruktor (= default kulcsszó).
- **Másoló** konstruktor
- **Csapda nélküli** mező gyors létrehozása
- **Csapdával ellátott** mező létrehozása

Annak érdekében, hogy könnyen elhelyezhessem az objektumot *map* adatstruktúrákba, definiáltam a *kisebb* (***operator<***) operátort, mely az *std::pair*-hez hasonlóan lexikografikus összehasonlítást[2] végez.

Az *egyenlőség* ***(operator==)*** operátor a *kisebb*hez hasonlóan az összehasonlítások megkönnyítése érdekében jött létre, az *értékadás* ***(operator=)*** operátor pedig egy gyorsabb alternatíva a **setter**ek mellett, amennyiben rögtön ismert, hogy milyen mezőt szeretnék értékül adni.

A ***reset()*** függvény csupán arra szolgál, hogy gyorsan vissza lehessen állítani a mező értékeit *TileType::Wall* és *TrapType::None* -ra.

[2] Szavak ábécé sorrendbe rendezéséhez hasonlóan szekvenciálisan összehasonlítja az első, majd a második elemet.

3.7.3. DungeonMap osztály

A **DungeonMap** osztály reprezentálja azt a pályát, amin a karakter mozog. Az objektumban megvalósítottam a ***reset(int)*** függvényt, ami képes az objektumot visszaállítani egy kezdő pozícióba annak érdekében, hogy ne legyen szükség új pálya generálásakor vagy betöltésekor az objektum újra konstruálására a **modell** szinten.

Az osztályban definiáltam a szükséges számú **gettereket**, továbbá egy **setter** függvényt és egy operátort. Az ***operator[](i)*** operátor visszaadja a letárolt *_map* mátrix i-edik vektorának referenciáját, így az operátor újabb alkalmazásával könnyedén lekérdezhető, illetve módosítható a pálya bármelyik indexű eleme. Az ***at(i,j)*** **getter** függvény a segítségével könnyen elérhető a pálya i-edik sorának j-edik elemének konstans referenciája.

A **modell** szintjén alkalmazott két legfontosabb publikus függvény a ***generateRandomMap(type),*** illetve a ***placeTraps(int)***, melyeket új pálya generáláshoz használok. A két függvényt az előbbi sorrendben érdemes használni.

- A ***generateRandomMap(type)*** függvény egy MapType típusú változót kér paraméternek, majd egy **switch** elágazás segítségével megfelelő típusú paraméterhez a megfelelő privát generátor függvényt hívom meg. Jelenleg egyetlen MapType van definiálva (Worm), melyhez a ***generateWormMap*** privát függvényt rendeltem, azonban jól látható, hogy ezzel a konstrukcióval egyszerűen és problémamentesen lehet **új pályatípusokat adni a programhoz.**
- A ***generateWormMap(sizeModifier)*** végzi a véletlen pálya tényleges generálását. A függvényben egy „giliszta” algoritmust alkalmaztam, mely **mindig biztosítja, hogy a végeredmény**nek kapott pálya minden pontja **bejárható legyen.** A módszer a következőképpen működik:
    1) A pálya alsó sorában egy kezdő koordináta generálása. Ez a koordináta lesz egyben az algoritmus kezdőpontja és a játékos karakter kezdőpontja. A generálás alatt ez a koordináta fogja reprezentálni a 2x2-es mező méretű *worm (giliszta)* bal alsó sarkának aktuális pozícióját.

2) Ettől a lépéstől kezdve egy ciklusba fut a függvény, mely annyiszor fut le, amennyi a *sizeModifier* paraméter értéke.
    - Egy random szám generálásának segítségével eldöntöm, hogy a giliszta melyik irányba lépjen tovább. A giliszta 30% eséllyel vált irányt, irányváltás esetén mindig az eddigi irányhoz képest balra vagy jobbra képes fordulni.
    - Amennyiben a kapott irány helytelen, vagyis a következő lépésben a giliszta a pálya szélére érkezne, vissza az előző lépésre. Helyes irány esetén megváltoztatom a giliszta aktuális pozícióját a megfelelő koordinátára.
    - Ezt követően a giliszta a megadott irányba „kiváj" két mező szélességű területet. Véletlenszerűen (75%-os eséllyel) a giliszta a megadott irányba nem 2 mező széles területet váj ki, hanem csak 1-et, ezáltal változatosabb és labirintus-szerűbb pályákat teremtve.
3) Ciklus végén beállítom a cél mezőt a legbalfelsőbb mezőre.

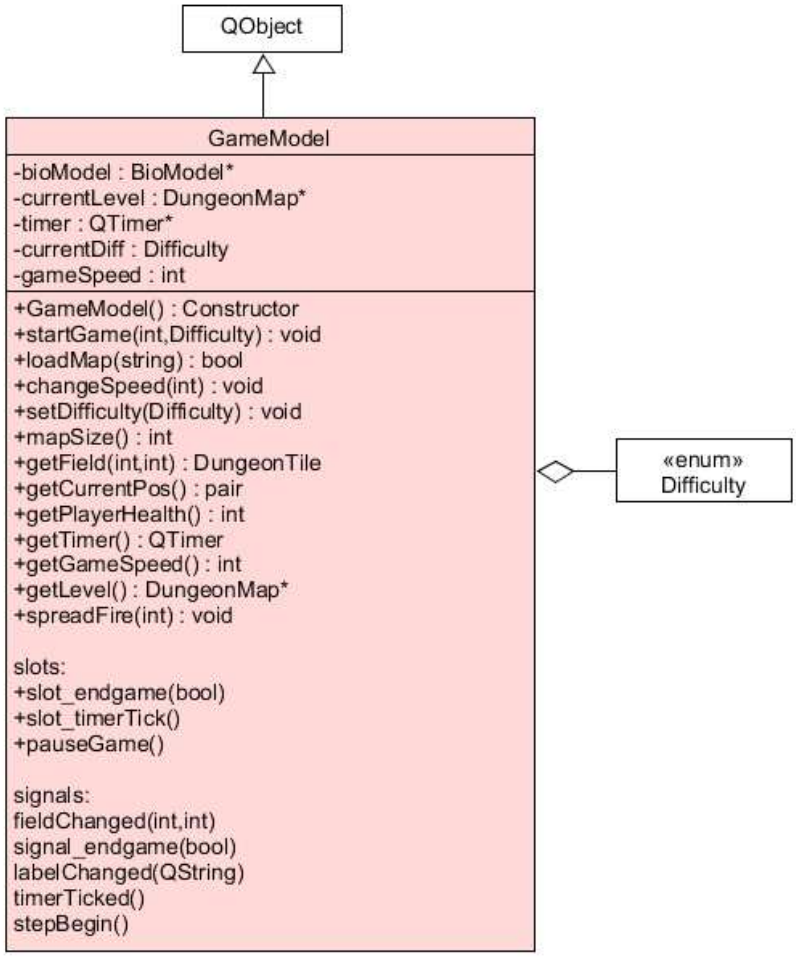
- A ***placeTraps(trapCount)*** függvény helyezi el egy pályára a megadott számmal arányos (de **nem** egyenlő számú) csapdát. Az algoritmus trapCount-szor lefutó ciklusban generál egy-egy random koordinátát. Amennyiben a generált koordinátán a pályán még nincs csapda és az bejárható terület, lehelyez egy véletlenszerűen választott csapdát. Ellenkező esetben a ciklus tovább lép. Ennek köszönhetően kisebb pályák nem lesznek „túlzsúfoltak" csapdákkal, hiszen ilyen esetekben nagyobb eséllyel fog a ciklus tovább lépni, mint csapdát rakni.

A **modell** szintjén használatos segédfüggvény a ***checkVicinity(x,y,tile,emptyFloors)***, ami egy (x,y) koordináta alatt elhelyezkedő mezőt határoló 9 pontot vizsgálja meg (a pálya szélein ez a szám értelemszerűen kevesebb). Az *emptyFloors* output paraméterbe elhelyezi azoknak a határoló mezőknek a koordinátáit, melyeken nincs csapda, visszatérési értékben pedig megadja, hogy a határoló mezők közül mennyi egyezik meg a *tile* paraméterben megadottal.

3.8. A Modell felső rétege (GameModel)

3.8.1. Felépítés

A **GameModel** osztály alkotja a **MainWidget** osztály „hátterét”. Ezen a szinten végzem a pálya felépítését és főbb módosításait. Ezen réteg látja el a **Nézet**et a szükséges információkkal, továbbá bizonyos esetekben szignál továbbításokkal **közvetítő** szerepet is játszik a **BioModel** és a **MainWidget** között. Felépítése a következő ábrán látható:



3.10. GameModel osztály szerkezete

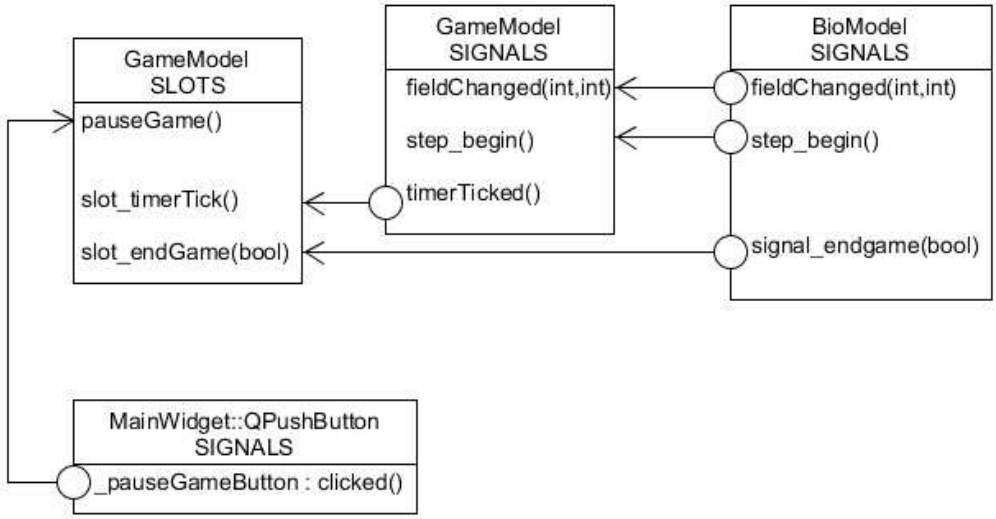
Megjegyzés: Az eddigiekhez hasonlóan a * jel *smart pointer*eket jelöl.

A **Difficulty** lehetséges értékei: {EASY, NORMAL, HARD}.

3.8.2. Szignál kapcsolatok

A **GameModel** számos helyről fogad szignálokat.

- A *_timer* adattag *timeout()* szignálját nem csak továbbítja a **BioModel** számára, hanem egy saját eseménykezelővel is rendelkezik.
- A **BioModel**-től érkező *fieldChanged(int,int), step_begin()* és *signal_endgame(bool)* szignálokat a **MainWidget** számára továbbítja.
- A **MainWidget**-ben található szüneteltetés gomb kattintás eseményét is lekezeli.



3.11. GameModel szignál kapcsolatai.

3.8.4. Az alkalmazás vezérlése

A felhasználó által megadott inputok a **nézet** rétegről a **modell** rétegbe továbbítódnak, ezt a réteget nevezhetjük a vezérlő rétegnek is.

A nézet felé történő kommunikációt két fronton valósítottam meg: *szignálok* (melyek nézet oldali lekezelését a 3.4. fejezetben részleteztem) és *publikus getterek* definiálásával. Ezen függvények segítségével a **nézet** számára lekérdezhető a jelenlegi *nehézség, pályaméret, karakter pozíció, játék sebessége*, továbbá maga a *_timer* objektum is elérhető. A *getField(int,int)* segítségével a pálya egy tetszőleges koordinátáján elhelyezkedő mező konstans referenciája is lekérdezhető.

A **nézet**ből meghívható vagy az onnan származó adatokat feldolgozó függvények a következők:

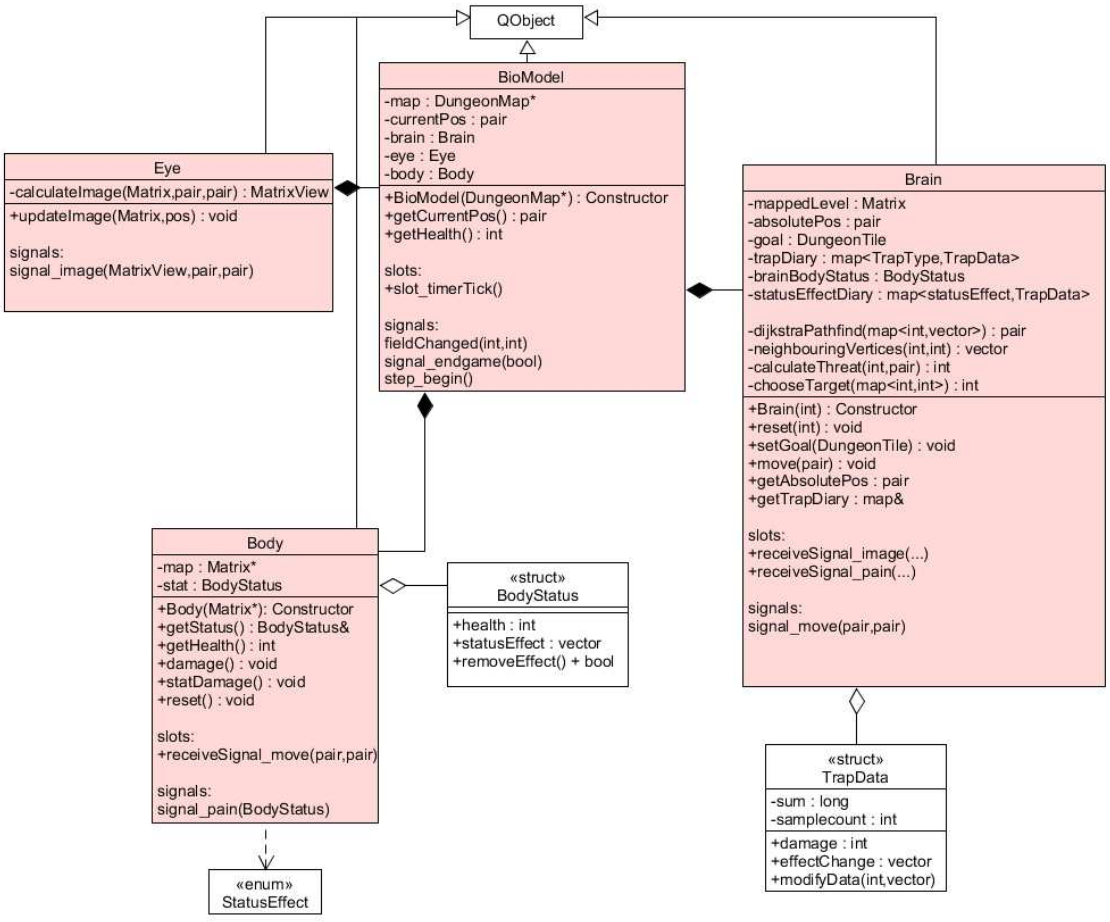
- ***startGame(mapSize, diff) :*** A *mapSize* paraméterben kapott méret alapján átméretezi és *reset*eli az aktuális pályát (*_currentLevel*). Ezt követően meghívom a *_currentLevel* pályageneráló függvényét, majd a *diff* változóban szereplő nehézség alapján felparaméterezett *placeTraps(int)* függvényt. Miután a pálya készen áll, meghívom a *_bioModel reset()* függvényét és újra indítom a *_timer*t az aktuálisan beállított sebességgel.
- ***loadMap(fname) :*** Az *fname* paraméterben átadott fájlelérési útvonallal és az aktuális pálya referenciájával felparaméterezve meghívja a DataAccess::loadMap függvényt.
- ***changeSpeed(speed) :*** A *speed* paraméterben megadott sebességre állítja a belső *_gameSpeed* változót, majd, amennyiben az időzítő nincs leállítva, újra indítja a *_timer*-t az új értékkel.
- ***setDifficulty(diff) :*** A bejövő paraméter értékére állítja a belső *_currentDiff* változót.
- ***pauseGame() :*** Amennyiben az időzítő szüneteltetve a *_gameSpeed* értékével felparaméterezve elindítja, ellenkező esetben megállítja azt.
- ***slot_timerTick() :*** A *_timer* objektum *timeout()* eseményére kötött eseménykezelő. Ennek a függvénynek a segítségével valósítom meg, hogy játékban minden lépés elején meghívódjon a *spreadFire* függvény, amennyiben a beállított nehézség meghaladja a könnyű fokozatot.
- ***slot_endGame() :*** A *bioModel* hasonló nevű szignáljára rákötött slot. Mielőtt továbbítaná a játék vége jelet a **nézet**nek, leállítja az időzítőt.
- ***spreadFire(percentage) :*** Ebben a metódusban valósítottam meg a **tűz terjedését a pályán.** Az algoritmus egy ciklusban végig iterál a pálya összes elemén, és ez *ignite* nevő vektorban gyűjti ki azon mezőknek a koordinátáit, melyeket „felgyújt” a függvény végén. A ciklusmag a következőképpen működik:

1) Amennyiben a vizsgált mező jelenleg nem „ég”, a ciklus tovább halad a következő elemre, ellenkező esetben 2).
2) Létrehoz egy vektor *vec*, illetve egy egész szám *fires* változót, majd a *_currentLevel checkVicinity(..)* függvényével kinyeri a vektorba az összes olyan határoló járható mezőt, ahol még nincs csapda, a *fires* változóba pedig lementi, hogy hány tűz mező határolja az aktuális pontot.
3) A következő lépésben a bejövő *percentage* változó segítségével egy *prob* változóba kiszámolja ezrelék pontossággal, hogy mekkora valószínűséggel fog terjedni a tűz a vizsgált pontból. Minél magasabb a *fires* szám, annál inkább csökkentjük ezt a valószínűséget annak érdekében, hogy ne terjedjen túl nagy ütemben a tűz.
4) Terjedés esetén kiválaszt egy véletlenszerű koordinátát a *vec* vektorban eltárolt szomszédos mezők közül, majd ezt hozzáfűzi az *ignite* vektorhoz.
5) Ciklus végén végig iterál az *ignite* vektoron, és minden benne található koordinátán levő mezőre meghívja a *setTrap(TileType::Fire)* metódust az említett paraméterrel.

3.9. A Biológiai Modell (BioModel)

3.9.1. Felépítés

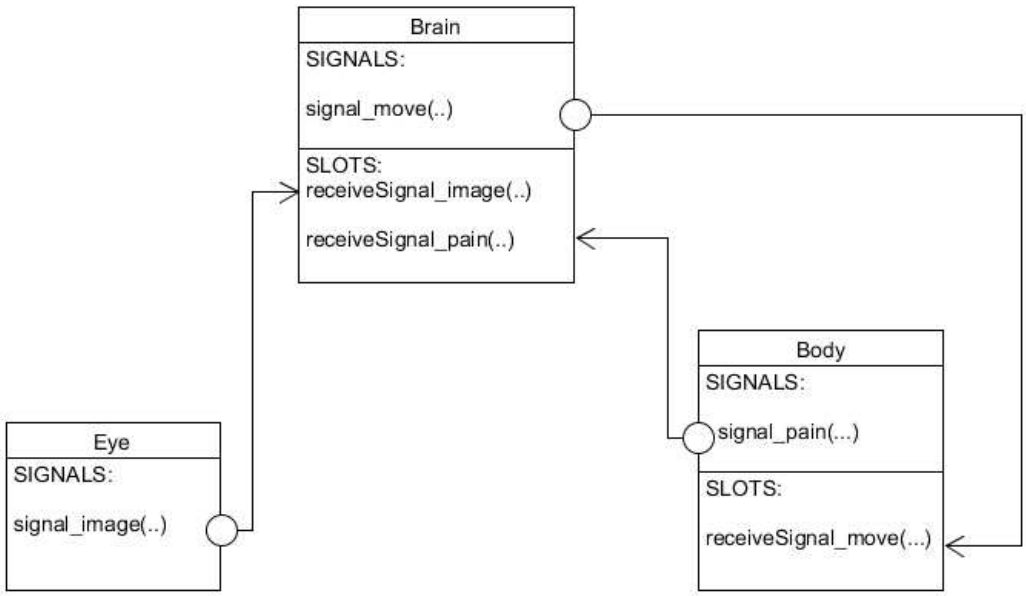
A **BioModel** osztály reprezentálja a játékban szereplő karaktert és foglalja magában a három „testrészét”: **Brain. Eye, Body**, melyek közti összeköttetésekért is felel. A **BioModel** értelmezhető egy „manager” osztályként, ami biztosítja, hogy karaktert reprezentáló testrészek összhangban működjenek egymással, továbbá az esetleges külső behatások helyes kivitelezése (például: sebzés) is a feladatkörébe tartozik. A biológiai modell részletes felépítése az alábbi ábrán látható:



3.12. BioModel struktúra felépítése

3.9.2. Szignál kapcsolatok

A különböző testrészek egymással való kommunikációját szignálok segítségével valósítottam meg. Ezen szignálok rendszerét a következő ábra szemlélteti:



3.13. Testrészek kommunikációja

3.9.3. A Szem (Eye)

A **szem** felel azért, hogy a karakter aktuális pozíciójának környezetét beazonosítsa, majd ezt a részmátrixot továbbítsa az **agy** számára. Ezt a részmátrixot a ***calculateImage(map,pos,relativePos)*** segítségével állítom elő.

- Az algoritmus a *map* input pályán számolja ki az eredményt. Kezdetben a *pos* koordinátától indulva szekvenciálisan a négy kardinális irányba megvizsgálja, hogy hány mezőre van a legközelebbi fal. A *pos* segítségével egy négyzetes részmátrix írható le.
- Létrehoztam egy *ret* nevű *MatrixView* típusú változót (konstans DungeonTile pointerekből álló mátrix), melybe az első lépésben kapott adatok alapján a *map* megfelelő részmátrixát beletöltöttem. A hatékonyság kedvéért választottam a MatrixView típust, így elég egy-egy mező referenciájával feltölteni a mátrixot.

- Az így kapott mátrixon megállapítottam a karakter relatív pozícióját és elmentem a *relativePos* input változóba (részmátrixon véve hányadik sor hányadik elemén helyezkedik el).
- Az előző lépésekben kapott részmátrix azonban még nem tökéletes, ugyanis jelenlegi állapotában a karakter „átlát a falakon”. Ezt úgy küszöböltem ki, hogy egy ciklus segítségével oszloponként végig futottam a mátrixon, majd az így nem elérhető mezők (azaz a karakter számára nem látható mezők) értékét *nullptr*-re állítottam.

A *calculateImage* segítségével kiszámolt részmátrixot és relatív pozíciót az **agy** számára a *signal_image* jel küldi el.

3.9.4. A Test (Body)

A **Body osztály** reprezentálja a karakter testét, azaz itt tároltam le a **BodyStatus** struktúra segítségével a karakter aktuális életét *(integer)* és a karaktert ért negatív effekteket *(std::multiset)*.

Ebben az osztályban valósítottam meg a **sebzés**ért felelős függvényeket is, továbbá a pályán való **mozgás** is itt van implementálva.

- ***damage(number,effects) :*** Ezt a függvényt a **BioModel** osztály hívja meg abban az esetben, hogyha a karakter csapdába lépett. A bejövő *number* paraméter értékével csökken a *_stat* adattagban található *health* értéke, emellett pedig a *statusEffects multiset*-be beszúrásra kerül az *effects* vektor összes eleme. A függvény végén a **test fájdalom** jelet küld az **agy** számára.
- ***statDamage(number, eff) :*** Ezt a függvényt a **BioModel** osztály hívja meg minden lépés elején. A bejövő *number* paraméter értékével csökken a *health,* de csak abban az esetben, hogyha az *eff StatusEffect* megtalálható a *statusEffects* halmazban. Ha a vizsgálat igaz, akkor kikerül a halmazból egy darab *eff* -el megegyező elem, megtörténik az élet csökkenés, a függvény a művelet végén pedig **fájdalom** szignált küld az **agy**nak.
- ***receiveSignal_move(dir,currentPos) :*** A **Toolbox** névtérben túlterhelt *operator+=* segítségével a *currentPos* output paraméter értékét inkrementáltam a *dir* értékével. Ezt követően hibaellenőrzést végeztem, arra, hogy az új pozíció a pályán egy bejárható mezőn helyezkedik el. Ha ez az

ellenőrzés hamissal térne vissza, akkor visszaállítanám a *currentPos* értékét az eredetire, azonban ez a hiba a jelenlegi programban nem fordulhat elő.

3.9.5. Az Agy (Brain)

Az **agy** egy központi vezérlő a többi testrész számára. Minden adat ebben az osztályban kerül feldolgozásra, és minden döntés itt születik meg. Ahhoz, hogy bemutassam az **agy** működését, először az itt tárolt adatokat kell ismertetnem.

- ***_mappedLevel :*** Az a részmátrix, amit a karakter ismer a pályából. Az útkeresési számítások is ezen objektum segítségével történnek. Mérete megegyezik a pálya méretével, azonban az eddig felfedezetlen mezők *TileType::Unknown* típusúak.
- ***_absolutePos :*** A *_mappedLevel* mátrixon mely koordinátát foglalja el a karakter.
- ***_goal :*** A ***setGoal(DungeonTile)*** setter segítségével beállított mező, ez jelzi a karakter végcélját.
- ***_trapDiary :*** *TrapType* típusokhoz ***TrapData*** objektumokat rendelő *unordered_map* (a kvázi konstans idejű elérés érdekében). A ***TrapData*** egy olyan objektum, melyben le van tárolva egy sebzés érték, továbbá egy vektor, melyben státusz effekt változások vannak elhelyezve. Az objektumot a *modifyData* tagfüggvénnyel lehet frissíteni, mely a privát *sum* és *sampleCount* adattagok segítségével az eddigi adatok és az új adat alapján egy átlagot számol. Ez az átlag lesz a *damage* adattag értéke. A függvény fel van készülve az esetleges *integer túlcsordulás*ra is.
- ***_brainBodyStatus :*** Egy *BodyStatus* objektum, ami a **test**ben találhatóhoz hasonlóan letárolja a karakter testének állapotát. Ez az objektum akkor frissül, amikor a **test**től **fájdalom** szignál érkezik az **agy**ba.
- ***_statusEffectDiary :*** *unordered_map,* ami egy adott *StatusEffect*-hez rendel egy *TrapData* értéket.

3.9.6. A szemtől kapott információ feldolgozása

Ahogy már korábban is ismertettem, a karakter kezdetben egy teljesen ismeretlen pályát érzékel a környezetéből, melyet a **szem**től fogadott részmátrixok segítségével épít fel. Ez az „épülő" pálya található a *_mappedLevel,* azaz *feltérképezett pálya* változóban. Ennek a mátrixnak a helyessége **kulcsfontosságú az útkereső algoritmus helyes működéséhez**. A szemtől kapott jelet a ***receiveSignal_image(image,absPos,relativePos)*** slot dolgozza fel, melyet a következőképpen valósítottam meg:

- Az *_absolutePos* változó értékét beállítja az *absPos* értékére.
- Kiszámolja a relatív és abszolút pozíció alapján, hogy a részmátrix bal felső eleme mely koordinátán helyezkedik el az eredeti pályán.
- Ettől a koordinátától kezdve, sorfolytonosan egyszerre haladva a *_mappedLevel* mátrixon, és a (0,0) koordinátától kezdve az *image* mátrixon, az *image* adott elemének értékét átmásolja a *_mappedLevel* megfelelő helyére. Amennyiben a vizsgált elem *nullptr,* úgy *TileType::Unknown* típusú mező kerül a feltérképezett pályára.

3.9.7. A fájdalom jel feldolgozása

A **test**től kapott fájdalom jel helyes feldolgozása szintén **kulcsfontosságú**, ugyanis az így szerzett információk alapján számolja ki az **agy** az ideális útvonalat, azaz ezek alapján rendel **súlyt** a gráf egy **pontjához.** A ***receiveSignal_pain(stat)*** függvényt a következőképpen valósítottam meg:

- A függvény alapelve, hogy az **agy** az alapján szerez információt arról, hogy mi történt a **testtel,** hogy összehasonlítja a test agyban tárolt státuszát a valódi, (*stat* paraméterben kapott) státusszal, megfigyelve a változásokat.
- Kezdetben kiszámolja a két státusz közötti életpont változást a *diff* paraméterbe, továbbá lekérdezi azt a mezőt a pályán (*standingPos pointer)*, ahol a karakter éppen áll.
- Amennyiben a karakter által elfoglalt mezőn csapda van, úgy egy ciklusban a bejövő adatok és a *_brainBodyStatus* alapján meghatározza azt a *StatusEffect* vektort, ami az összes olyan effektet tartalmazza, amiktől eddig még nem szenvedett a karakter. A kiszámolt *diff* sebzéssel, *StatusEffect*-ekkel és a mezőn lévő csapdával bekerül vagy frissül egy bejegyzés a *_trapDiary*-ban.

- Ezt követően frissül az agyban letárolt **test** státusz a paraméterben kapott státuszra.
- Amennyiben a karakter üres mezőn áll, de mégis sebzés érte, az azt jelenti, hogy valamilyen *StatusEffect* hatására történt ez, ezért a *_statusEffectDiary* frissül a *diff* értékkel.

3.9.8. Az útkeresés

Az **agy** a ***move(currentPos)*** függvény hívásával képes meghatározni az ideális út következő lépését. Az **ideális utat** a következőképpen **definiáltam**: Az a legkevesebb költségű út, amelynek célállomása egy még **felfedezetlen** mező, vagy pedig a **cél** mező, abban az esetben, ha az elérhető a karakter jelenlegi állapotával. Ehhez a jól ismert **Dijkstra algoritmust** használtam, aminek a pszeudo-kódja az alább látható:

```
1:  function Dijkstra(Graph, source):
2:      for each vertex v in Graph:
3:          dist[v] := infinity
4:          previous[v] := undefined
5:      dist[source] := 0
6:      Q := the set of all nodes in Graph
7:      while Q is not empty:
8:          u := node in Q with smallest dist[ ]
9:          remove u from Q
10:         for each neighbor v of u:
11:             alt := dist[u] + dist_between(u, v)
12:             if alt < dist[v]
13:                 dist[v] := alt
14:                 previous[v] := u
15:     return previous[ ]
```

3.14. A Dijkstra algoritmus pszeudo-kódja[2]

Megjegyzés: A kódban egy koordinátát a 3.2. fejezetben ismertetett *makeVertex* függvény alkalmazásával alakítok át egy nemnegatív egész számmá, ami egyértelműen meghatároz egy gráfcsúcsot.

A 10. sorban látható pszeudo-kódból észrevehető, hogy szükség van egy olyan adatstruktúrára, ami egy csúcshoz hozzárendeli az összes **szomszédját**, ezért az előkészületek során előállítottam a *neighbours* nevű *std::map* típusú objektumot.

Mivel a legtöbb esetben a feltérképezett pálya (négyzetrácsgráf) nagy része **használhatatlan**, vagyis olyan pontok, amik elérhetetlenek a karakter számára, ezért nem szerettem volna, hogy ezek a csúcsok feleslegesen bekerüljenek a szomszédsági *map*-be. E célból hoztam létre a ***neighbouringVertices(i,j)*** segédfüggvényt, ami visszatér a *_mappedLevel[i][j]* elemének azon szomszédjaival, melyek valóban bejárható csúcsok. A **bejárható csúcsokat** a következőképp definiáltam: vagy **bejárható** mező, vagy olyan **ismeretlen** mező, amit határol legalább egy bejárható mező.

A szomszédsági *map* elkészültével meghívható a ***dijkstraPathfind(neighbours)*** függvény. A pszeudo-kódtól eltérően ebben az esetben nem szükséges külön beadni a forrás csúcsot, hiszen ez nem más, mint a karakter jelenlegi pozíciója. A következő lépésben inicializáltam a *dist* és *previous unordered_map* változókat. Az előbbi egy nemnegatív egész számhoz (**csúcs**) rendel *long int*-et (**költség**), az utóbbi pedig egy nemnegatív egész számot egy másik nemnegatív egész számhoz (adott csúcsba melyik csúcsból juthatunk el). A *dist* kezdetben minden csúcshoz *LONG_MAX* értéket rendel, amit a kódban végtelennek értelmeztem, a *previous* pedig *0*-t, amit undefined-ként értelmeztem.

Első lépésben a csúcsba vezető költséget lenulláztam, majd létrehoztam egy *Vertex* típusú (bővebben: 3.2. fejezet) elemekből álló vektort, aminek a pszeudo-kódban látható *Q* nevet adtam, a csúcsokhoz tartozó *val* pointert pedig a *dist* megfelelő csúcsához rendelt költség referenciájára állítottam be. Erre azért volt szükség, hogy amikor a távolság map-ben változik egy csúcshoz tartozó költség, úgy a Q vektorban ne kelljen ezt külön változtatni.

Következő lépésben egy ciklus egymás után kiválasztja a Q vektorból a legkisebb költséggel rendelkező súlyokat, a ciklusmag végén pedig eltávolítja azokat. A minimum kiválasztást az *std::min_element* függvény segítségével végzem. A ciklus magja:

- Minimum kiválasztást követően egy belső ciklus következik, mely az aktuálisan kiválasztott csúcs szomszédjain iterál végig.
- A kód az ábra 11. sorában leírtaktól abban tér el, hogy egy segédfüggvény segítségével számolja ki az *alt* változóba a két csúcs közti kumulatív költséget. A ***calculateThreat(prevCost, destination)*** függvény a költség kiszámításához az alábbiakat veszi figyelembe:

    - A karakter jelenlegi *StatusEffect*jei
    - A karakter jelenlegi életpontja (*health*)
    - A *destination* mezőn elhelyezett csapdák alapján az agyban eddig letárolt *_statusEffectDiary*-ben és a *_trapDiary*-ben található adatok.
- Költség számításnál a kezdő költség 1, a későbbi vizsgálatok során számolt költségek pedig 10^3 nagyságrendűek. Erre azért volt szükség, mert nem szerettem volna, ha egy 10-es nagyságrendű utat is „fenyegetésnek” tekintsen a karakter, hiszen az ilyen úton biztosan nincs veszély, azonban ebben az esetben 0 sem lehetett a számolt költség, hisz ekkor nem lehetne különbséget tenni két veszélytelen út hossza között.
- Amennyiben a számolt költség alapján azt állapítom meg, hogy a karakter nem élné túl az ide vezető utat, úgy a költséget végtelenre állítom. Természetesen hibakezelést ebben a függvényben is végeztem, hiszen többször LONG_MAX számokkal is dolgozom, melyekkel a jelenlegi kontextusban nincs értelme műveleteket végezni.
- Költség kiszámítását követően a ciklus további rész a pszeudo-kódnak megfelelően fut le.

A ciklust követően a *dist* távolságokat tartalmazó map teljesen fel van töltve. Ekkor a következő lépés ezen utak közül kiválasztani a megfelelőt. A ***chooseTarget(dist)*** függvény egy **feltételes minimum kiválasztás** segítségével kiválasztja azt a *Unknown* típusú mezőt, melyhez a legkevesebb költség van rendelve, **vagy** pedig a cél mezőt, amennyiben nem végtelen van hozzá rendelve. Ha minden lehetséges út költsége végtelen az azt jelenti, hogy a karakter egyik célpontot se éri el anélkül, hogy meghaljon, ekkor a függvény a _currentPos-al tér vissza, jelezve azt, hogy nincs hová menni.

Célpontválasztást követően a *previous* segítségével kiszámolom, hogy a határoló csúcsok közül melyik lesz a következő lépés célpontja, majd a függvény visszatér a két koordináta (jelenlegi pozíció és cél pozíció) különbségével, amit pedig a *move()* függvény *signal_move* jel segítségével elküld a **test** számára.

3.9.9. Egy lépés a BioModel szintjén

A BioModel minden komponensének működését részletesen ismertettem, így nem maradt más, mint demonstrálni, hogyan is zajlik egy teljes lépés és a gyakorlatban hogyan valósul meg a testrészek közti kommunikáció.

1) TimerTick esemény bekövetkezik
2) BioModel megvizsgálja, hogy milyen mezőn áll a karakter. Amennyiben a karakter a célon áll, ***signal_endgame(true)*** (játék vége) jelet küld a **GameModel** számára. Ha a karakter csapdán áll, akkor meghívja a **test** *damage* műveletét egy véletlenszerűen generált számmal és a megfelelő *StatusEffect* vektorral.
3) Meghívódik a **test** *statDamage* metódusa az adott *StatusEffect*-tel és egy random számmal.
4) ***step_begin*** szignál küldése a **GameModel**-nek, ahol ezt követően bekövetkezik a pályán lévő tűz terjedése.
5) Amennyiben a karakter élete 0 alá csökkent, úgy ***signal_endgame(false)*** küldése.
6) A **szem** „kinyitása”, aminek hatására az agy megkapja az éppen megfigyelt részmátrixot, és ezzel frissíti a feltérképezett pályát.
7) Az **agy** *move* mdetódusának hívása, aminek a végén megváltozik a *_currentPos* értéke.
8) Amennyiben az új pozíció ugyanaz, mint a régi, vagy nem mozgott a karakter, ***signal_endgame(false)*** küldése.
9) ***fieldChanged*** szignál küldése a modellnek, mivel változott a karakter pozíciója.

3.10. Tesztelés

3.10.1. Unit Testek

A projekthez készítettem egy **Qt Test** keretrendszerben megírt projektet, melynek forráskódja a *tst_testcase.cpp* fájlban található. Az alábbi teszteket futtattam:

- **MapGenerationTest() :** Ellenőriztem, hogy a *generateRandomMap* segítségével generált pályák mindig bejárhatók legyenek a kezdő pozíciótól.
- **dataAccessTest() :** Ellenőriztem, hogy
  - A *saveMap* függvény csak a helyesen elkészített pályákat menti el
  - A *loadMap* függvény jól betölt-e egy helyesen elmentett pályát
- **topLevelTest() :** Ellenőriztem, hogy a modell setter függvényei helyesen működnek-e, továbbá azt, hogy pályabetöltés után elindul-e a játék.
- **bioModelTest() :** Ellenőriztem, hogy a bioModelben lévő slot_timerTick() függvény hatására tovább halad-e a játék, majd azt, hogy a karakter egy egyszerű pályán beér-e a célba, továbbá azt, hogy a csapdák sebzik-e a karaktert.

3.10.2. Fejlesztéssel párhuzamos tesztek

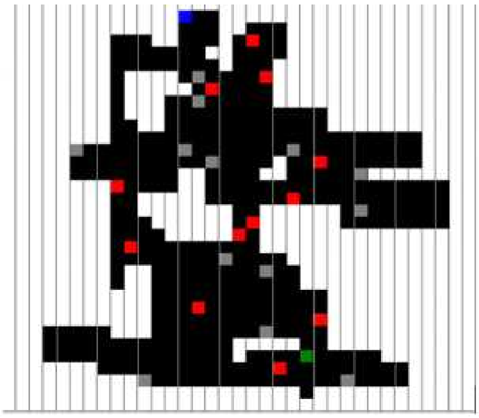
Korábban említett módon a *toolbox.hpp* headerben definiált makrókat felhasználva a fejlesztés során is végeztem teszteket. A program véletlenszerű mivolta miatt ezeket a teszteket nagyon nehéz lett volna automatizálni, ezért saját kezűleg ellenőriztem a helyes működést.

A **DEBUG** makró a fejlesztői környezetben nyújtott hasznos információkat, a **LOG** makró pedig a megadott fájlokba mentette a szükséges adatokat. Az alábbiakban egy-egy példa látható ezek használatára.

```
updateImage :[DEBUG]: Eye updating image.
updateImage :[DEBUG]: Emitting image signal.
receiveSignal_image :[DEBUG]: Brain received image signal.
dijkstraPathfind :[DEBUG]: Brain using Dijkstra Pathfind.
move :[DEBUG]: Brain sending move signal to direction: std::pair(-1,0)
receiveSignal_move :[DEBUG]: Body received move signal.
[ STEP ]: It took me 0.0053128 seconds.
```

3.15. Egy helyesen lefutó lépés DEBUG üzenetei.

```
Position: 7 16
1 2 2 2 2 0 0 0 0 0 0 2 2 2 2 2 1 1 2 2 2 2 2 2 1
1 2 2 2 2 0 0 0 0 0 0 2 2 2 2 2 0 0 2 2 2 2 2 2 1
1 2 2 2 2 0 0 0 0 0 0 2 2 2 2 2 0 0 2 2 2 2 2 2 1
1 1 2 2 2 0 0 0 0 0 0 2 2 2 2 2 0 0 2 2 2 2 2 2 1
0 0 1 1 1 0 0 0 0 0 0 2 2 2 2 2 0 0 2 2 2 2 2 2 1
0 0 0 0 0 0 0 0 0 0 0 2 1 1 1 1 0 0 1 1 1 1 2 2 1
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1
1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 1
```



3.16. ***bal****: A szem által látott részmátrix*
jobb*: tényleges pálya, ahol a karakter már egy lépéssel előbbre jár, mint amit a szem lát*
lenn*: az agyban tárolt feltérképezett pálya*
magyarázat*:0 – talaj, 1 – fal, 2 - ismeretlen*

```
Position: 31 22
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 1 2 2 1 2 0 0 0 0 0 0 2 2 2 2 0 1 1 2 2 2 2 2 2 1 2 2 2 2
2 2 2 2 2 2 1 2 2 1 2 0 0 0 0 0 0 2 2 2 2 0 0 0 2 2 2 2 2 2 1 2 2 2 2
2 2 2 2 2 2 1 2 2 1 2 0 0 0 0 0 0 2 2 2 2 0 0 0 2 2 2 2 2 2 1 2 2 2 2
2 2 2 2 2 2 1 1 2 1 2 0 0 0 0 0 0 2 2 2 2 0 0 0 2 2 2 2 2 2 1 2 2 2 2
2 2 2 2 2 2 0 0 1 1 1 0 0 0 0 0 0 2 2 2 2 0 0 0 2 2 2 2 2 2 1 2 2 2 2
2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 2 1 1 1 1 0 0 1 1 1 1 2 2 1 2 2 2 2
2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 2 2 2 2
2 2 2 2 2 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 2 2 2
2 2 2 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 2 2 2
2 2 2 2 2 2 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 2 2 2 2
2 2 2 2 2 2 1 2 2 1 2 2 2 2 2 2 2 2 2 2 2 1 1 1 2 2 2 2 2 2 1 2 2 2 2
```

Összefoglalás

Összességében elmondható, hogy az eredeti terveket jól megvalósító, azokon felül teljesítő programot készítettem a szakdolgozat kereteiben. A tervezés legkorábbi szakaszában még nem fogalmazódott meg bennem a biológiai modell ötlete, azonban szerettem volna a programban szereplő tanítást és útkeresést egy érdekes köntösbe borítani. Ekkor született meg a program modelljének jelenlegi szerkezetének a terve. Tervezést követően egyértelművé vált számomra, hogy milyen keretrendszerben fogom készíteni az alkalmazást, ezt követően pedig egyszerre csak egy-egy kisebb részre fókuszálva, azt tökéletesítve valósítottam meg az implementációt. Ennek a módszernek köszönhetően a fejlesztési fázis korábbi szakaszában írt munkáimat csak nagyon ritkán kellett módosítani.

Munkám során nagy segítségemre volt a C++ hivatalos dokumentációja[3], mely segítségével a nyelvvel kapcsolatos bármilyen információt könnyen és gyorsan elérhettem. A Qt-ben szereplő objektumok felépítéséről, működéséről és helyes alkalmazásáról a Qt 5.12-es dokumentációjában[4] olvastam. A fejlesztés végeztével a Qt projektem Windows operációs rendszerre történő kitelepítésének módszerét és az ezzel kapcsolatos hibák megoldását a Qt Wiki[5] elolvasásával tanultam meg.

Irodalomjegyzék

[1] https://en.wikipedia.org/wiki/Lattice_graph - 2019.05.12.

[2] http://www.gitta.info/Accessibiliti/en/html/Dijkstra_learningObject1.html - 2019.05.

[3] http://www.cplusplus.com/ - 2019.02-től a program befejezéséig

[4] https://doc.qt.io/qt-5/ - 2019.02-től a program befejezéséig

[5] https://wiki.qt.io/Deploy_an_Application_on_Windows – 2019.05.11.